AFRL-IF-RS-TR-2005-342
Final Technical Report
September 2005

AUTOMATED DIVERSITY IN COMPUTER SYSTEMS

University of New Mexico

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. N286

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-342 has been reviewed and is approved for publication

APPROVED: /s/

NELSON P. ROBINSON
Project Engineer

FOR THE DIRECTOR: /s/

WARREN H. DEBANY, JR., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	SEPTEMBER 2005	Final Jun 02 – Mar 03

4. TITLE AND SUBTITLE
AUTOMATED DIVERSITY IN COMPUTER SYSTEMS

5. FUNDING NUMBERS
C - F30602-02-1-0146
PE - 61101E/62301E
PR - N286
TA - A6
WU - 10

6. AUTHOR(S)
Stephanie Forrest

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of New Mexico
Student Services Center
Albuquerque New Mexico 87131

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFGB
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2005-342

11. SUPPLEMENTARY NOTES
AFRL Project Engineer: Nelson P. Robinson/IFGB/(315) 330-4110/ Nelson.Robinson@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***
Attackers penetrate a large number of computers by exploiting common vulnerabilities. The objective of this effort is to address this internet-wide weakness by introducing diversity into computers so that a successful attack on one computer does not necessarily work on another one, even though it may be running identical software.

14. SUBJECT TERMS
Diversity, Vulnerabilities

15. NUMBER OF PAGES
99

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Appendixes

Key Idea:

Introducing diversity into computers---even those running identical software---so that successful attacks on one computer do not necessarily work on others. Diversity is one aspect of the kind of adaptive and robust methods used routinely in biological systems. Other adaptations beyond diversity were developed under this effort in the same spirit of increasing the resilience of our computing infrastructure.

Summary of results:

1. We explored three potential diversity mechanisms: Dynamic translation of machine code (to defend against code-injection), randomizing the system-call interface (to defend against code-injection attacks), and evolving diverse implementations of TCP resource management policies (to defend against Denial of Service (DoS) attacks). Of these, we focused primarily on the first mechanism---dynamic translation. Dynamic translation of machine code is used to achieve Randomized Instruction Set Emulation (RISE), which thwarts malicious code injection attacks by making injected code appear random and thus illegal to the native processor.

2. We developed a prototype implementation of machine code randomization, called RISE (randomized instruction set emulation). RISE is available under GPL licensing from: http://www.cs.unm.edu/~gbarrant. We tested RISE's performance at stopping attacks using the Core Impact testing software. We conducted extensive experiments, both to test the effectiveness of RISE at stopping attacks and to test the safety of executing random sequences of instructions (during a code-injection attack, the attack code is effectively randomized and we wanted to know how quickly and with how much certainty such randomized code would fail).

3. We also continued the development of a dynamic translation tool (SIND) to facilitate dynamic translation diversity on RISC platforms. These results were reported in Trek Palmer's MS Thesis.

4. We performed experimental validation and probabilistic analysis of RISE, for both RISC and CISC architectures, demonstrating and explaining the efficacy of the RISE approach. RISE succeeds in converting a very high percentage of malicious code injection attacks into no more than just barely detectable denials of service. In other words, its affect on a particular computer or network of computers was minimal.

5. We also experimented with diversification of TCP resource management policies to prevent a form of resonance present in current networks. Hosts implement TCP with uniform policies and parameters, even when the uniformity is not necessary for correctness. A class of network Denial of Service attacks is able to drive such networks into pathological “resonance” modes in which the network throughput is degraded even in the absence of true load. Diversification of TCP resource

management policies can reduce the likelihood that such resonance modes exist or can be discovered by the attacker. We developed a prototype implementation which randomizes four TCP protocol parameters. Evaluation of network performance uses trace-based simulation. This work is promising but not fully mature.

6. Collaboration with Carnegie Mellon Univ. (CMU), Prof Mike Reiter. We established collaboration with Mike Reiter of CMU that has led to a follow-on jointly funded research project extending the results obtained under this grant.

7. Products:

RISE prototype software described above.

Appendixes

Appendix A - G. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefnaovic, and D. Zovi, "***Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks***", ACM Conference on Computer and Communications Security (2004).

NOTES: Barrantes, Ackley, and Stefanovic attended the conference and presented this paper. This paper was selected to be expanded into a journal-length article for publication in ACM TISSEC (attachment #3 below).

Appendix B - T. Palmer: "***Design and Implementation of SIND, a Dynamic Binary Translator***", UNM MS Thesis, December 2003.

Appendix C - G.Barrantes, D. Ackley, S. Forrest, and D. Stefanovic, "***Randomized Instruction Set Emulation"*** ACM Transactions on Information Systems Security (TISSEC), May 2004.

NOTE: This paper was written after end of grant, but was based on results obtained during time of this grant.

Appendix A - Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks

Elena Gabriela Barrantes
University of New Mexico
gbarrant@cs.unm.edu

David H. Ackley
University of New Mexico
ackley@cs.unm.edu

Stephanie Forrest
University of New Mexico
forrest@cs.unm.edu

Trek S. Palmer *
University of New Mexico
tpalmer@cs.unm.edu

Darko Stefanović
University of New Mexico
darko@cs.unm.edu

Dino Dai Zovi †
University of New Mexico
ddaizovi@atstake.com

ABSTRACT

Binary code injection into an executing program is a common form of attack. Most current defenses against this form of attack use a 'guard all doors' strategy, trying to block the avenues by which execution can be diverted. We describe a complementary method of protection, which disrupts foreign code execution regardless of how the code is injected. A unique and private machine instruction set for each executing program would make it difficult for an outsider to design binary attack code against that program and impossible to use the same binary attack code against multiple machines. As a proof of concept, we describe a *randomized instruction set emulator* (RISE), based on the open-source Valgrind x86-to-x86 binary translator. The prototype disrupts binary code injection attacks against a program without requiring its recompilation, linking, or access to source code. The paper describes the RISE implementation and its limitations, gives evidence demonstrating that RISE defeats common attacks, considers how the dense x86 instruction set affects the method, and discusses potential extensions of the idea.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.3.4 [**Programming Languages**]: Processors

General Terms

Security, Languages

Keywords

Automated Diversity, Security, Emulation, Language Randomization, Obfuscation, Information Hiding

*Currently at the University of Massachusetts, Amherst.

†Currently at @stake Inc.

CCS'03, October 27–30, 2003, Washington, DC, USA.

1. INTRODUCTION

Standardized interfaces between software and hardware are a double-edged sword. On the one hand, they lead to huge productivity improvements through independent development and optimization of hardware and software. But, they also allow a single attack code designed against an exploitable flaw to gain control of thousands or millions of standardized systems. One approach to controlling this form of attack is to 'destandardize' the protected system in an externally unobservable way, so that an outside attacker either cannot easily obtain the information needed to craft the attack or must manually regenerate the attack once for each new attack instance. Techniques that take this approach include obfuscation, information hiding, and automated diversity.

In the case of binary code injection, many defense techniques act to block known routes by which foreign code is placed into the execution path of a program. For example, stack defense mechanisms [20, 37] that protect return addresses defeat large classes of buffer overflow attacks; separate techniques [18, 32] defeat buffer overflows in other write-accessible parts of address space. Attacks such as 'return into libc' [30] avoid injecting any executable code at all, instead altering only data and addresses so that code already existing in the program is subverted to execute the attack; defense techniques like address obfuscation [16, 14, 32] counter by hiding and/or randomizing existing code locations.

In addition to such 'perimeter defense' techniques aimed at specific attack vectors, a secret destandardization of the executing code itself offers a complementary and quite general method of protection. With such instruction set obfuscation, each program (or process, or machine, or other unit of machine code execution) has a different and secret instruction set. If the number of possible instruction sets is large and externally unobservable, the cost of developing an attack from the outside is greatly increased, and different attacks must be crafted for each protected system.

In this paper we describe *randomized instruction set emulation* (RISE), an instruction set obfuscation technique implemented at the machine emulator level. Each byte of protected code in the program is individually scrambled using pseudorandom numbers seeded with a random key that is unique to each program execution. With the scrambling constants it is trivial to transform the obfuscated code back to normal instructions executable on the physical machine, but without knowledge of the key it is infeasible to produce even a short code sequence that implements any given behavior. Foreign binary code that reaches the path of execution will be descrambled without ever having been correctly scrambled, pro-

ducing essentially random bits that will usually crash the program under attack.

1.1 Threat model

RISE does not address, let alone solve, all possible security problems or even all possible attacks via communications networks. Our specific threat model is *binary code injection from the network into an executing program*. This includes many real world attacks, but explicitly excludes others, such as macro viruses that involve injection of something other than binary code, or the 'data injection' attacks mentioned above that do not rely on machine code. We assume that attacks arrive via network communications and that the contents of local disks are therefore trustworthy before an attack has occurred.

Our threat model is related to, but distinct from, other models used to characterize buffer overflow attacks [21, 18], so it is important to compare and contrast the approaches. Our threat model includes any attack in which native code is injected into a running binary, including misallocated malloc headers, footer tags [2], and format string attacks that can write a byte to arbitrary memory locations without actually overflowing a buffer [31]. RISE will protect against injected code arriving by any of these methods. On the other hand, other buffer overflow defenses, such as the address obfuscation mentioned earlier, can prevent attacks that are specifically excluded from our code-based threat model. RISE provides no defense against data-only attacks, which can range from the modification of jump addresses and parameters to call an existing library function (such as the family of return-into-libc attacks [30]) to the modification of password files or other critical information (for example, a privilege escalation as in [4]).

We envision the relatively general code-based mechanism of RISE being used in conjunction with more specific data- and address-based mechanisms to provide deeper, more principled, and more robust defenses against both known and unknown attacks.

1.2 Overview

In this paper we present a proof-of-concept RISE system, building randomized instruction set support into a version of the Valgrind x86-to-x86 binary translator [36]. In Section 2 we describe a *randomizing loader* for Valgrind that scrambles code sequences loaded into emulator memory from the local disk using a hidden random key. Then, during Valgrind's emulated instruction fetch cycle, fetched instructions are unscrambled, yielding unaltered x86 machine code runnable on the physical machine. The RISE design makes few demands on the supporting emulator and could be easily ported to any binary-to-binary translator for which source code is available.

In Section 3 we present our experimental results. We have found that RISE is successful in preventing code injection attacks, both synthetic and real, as described in Section 3.1. Section 3.2 analyzes the potential problem of creating valid instructions with the randomization given the dense x86 instruction set and Section 3.3 comments on performance issues.

When binary attack code, arriving over the network, exploits a bug and manages to interpose itself into the emulator execution path, the injected code will not have been scrambled by the loader. Consequently, when the attack code is fetched and unscrambled by the emulated instruction unit, it will appear as an essentially random string of bits. Despite the density of the x86 instruction set, we present data suggesting that the vast majority of random code sequences will encounter an address fault or illegal instruction quickly, aborting the program. Thus with RISE, an attack that would otherwise take control of a program is downgraded into a denial-of-service attack against the exploitable program. Regardless of what flaw is exploited in a protected program—whether well-known or entirely novel—the network binary code injection attack will fail with very high probability.

Section 4 summarizes related work, and Section 5 concludes with a general discussion.

2. TECHNICAL APPROACH AND IMPLEMENTATION

This section describes the prototype implementation of RISE using Valgrind [36] for the Intel x86 architecture. The RISE strategy is to provide each program copy its own unique and private instruction set. To do this, we consider what is the most appropriate machine abstraction level, how to scramble and descramble instructions, when to apply the randomization and when to descramble, and how to protect interpreter data. We also describe idiosyncrasies of Valgrind that affected the implementation.

2.1 Machine abstraction level

The native instruction set of a machine is a promising computational level for automated diversification. Since all computer functionality can be expressed in machine code, it is a desirable level to attack and protect. Also, with a network-based threat model, all legitimately executing machine code comes from the local disks, providing a clear trust boundary. By contrast, a Javascript interpreter in a web browser would be a poor candidate for this approach, because most Javascript code arrives over the network without firm trust boundaries between more and less legitimate code sequences.

A drawback of native instruction sets is that they are traditionally physically encoded and not readily modifiable. RISE therefore works at an intermediate level, using software that performs binary-to-binary code translation. The performance impact of such tools can be minimal [11, 15]. Indeed, binary-to-binary translators sometimes improve performance compared to running the programs directly on the native hardware [11]. For ease of research and dissemination, we selected an open-source system, Valgrind [36], as the basis for our demonstration implementation.

Although Valgrind is billed primarily as a tool for detecting memory leaks and other program errors, it contains a complete x86-to-x86 binary translator. The primary drawback of Valgrind is that it is very slow, largely due to its extensive access checking. However, the additional slowdown imposed by adding RISE to Valgrind is modest (see Section 3), and we are optimistic that porting RISE to a more performance-oriented emulator will yield a fully practical code defense.

2.2 Instruction set randomization

Instruction set randomization could be as radical as developing a new set of opcodes, instruction layouts, and a key-based toolchain capable of generating the randomized binary code. And, it could take place at many points in the compilation-to-execution spectrum. Although performing randomization early could help distinguish code from data, it would require a full compilation environment on every machine, and recompiled randomized programs would likely have one fixed key indefinitely. RISE randomizes as late as possible in the process, scrambling each byte of the trusted code as it is loaded into the emulator, and then unscrambling it before execution by the virtual machine. Deferring the randomization to load time makes it possible to scramble and load existing files in the Executable and Linking Format (ELF)[38] directly, without recompilation or source code, provided we can reliably distinguish code from data in the ELF file format.

The unscrambling process needs to be fast, and the scrambling process must be as hard as possible for an outsider to deduce. Our current approach is to generate at load time a pseudo-random sequence the length of the overall program text using the Linux `/dev/urandom` device [39], which uses a secret pool of true randomness to seed a pseudo-random stream generated by feedback through SHA1 hashing. The resulting bytes are simply XORed with the instruction bytes to scramble and unscramble them. If the underlying truly random key is long enough, and as long as it is infeasible to invert SHA1 [35], then we can have confidence that an attacker could not break the entire sequence. We return to the issue of how secure the RISE encoding in Section 5.

2.3 Design decisions

Two important aspects of the RISE implementation are how it handles shared libraries and how it protects the plaintext executable.

Much of the code executed by modern programs resides in shared libraries. This form of code sharing can significantly reduce the effect of the diversification, as processes must use the same instruction set as the libraries they require. When our load-time randomization mechanism writes to memory that belongs to shared objects, the Operating System does a copy-on-write, and a private copy of the scrambled code is stored in the virtual memory of the process. This significantly increases memory requirements, but increases interprocess diversity and avoids having the plaintext code mapped in the protected processes' memory.

Protecting the plaintext instructions inside Valgrind is a second concern. As Valgrind simulates the operation of the CPU, during the fetch cycle when the next byte(s) are read from program memory, RISE intercepts the bytes and unscrambles them; the scrambled code in memory is never modified. Eventually, however, a plaintext piece of the program (semantically equivalent to a basic block) is written to Valgrind's cache. From a security point of view, it would be best to separate the RISE address space completely from the protected program address space, so that the plaintext is inaccessible from the vulnerable program, but as a practical matter this would slow down emulator data accesses to an extreme and unacceptable degree. For efficiency, the RISE interpreter is best located in the same address space as the target binary, but of course this introduces some security concerns. A RISE-aware attacker could aim to inject code into a RISE data area, rather than that of the vulnerable process. This is a problem because the cache cannot be encrypted. To protect it, cache pages are kept as read and execute only. When a new translated block is ready to be written to the cache, we mark the affected pages as writable, execute the write action, and return them to their original non-writable permissions.

2.4 Implementation issues

An emulator needs to create a clear boundary between itself and the process to be emulated. In particular, the emulator should not use the same shared libraries as the process being emulated. Valgrind deals with this issue by adding its own implementation of any library function it requires using a local name, for example, `VGplain_printf(...)` instead of `printf(...)`. However, we discovered that Valgrind occasionally jumped into the target binary to execute low-level functions (e.g., `__umoddi` and `__udivdi`). When that happened, the processor attempted to execute instructions that had been scrambled for the emulated process, causing Valgrind to abort. Although this was irritating, it did demonstrate the robustness of the RISE approach in that these latent 'boundary crossings' were immediately detected. We worked around these dangling unresolved references by adding more local functions and renaming affected symbols with local names (e.g., rise_umoddi(...) instead of '%' (the modulo operator)).

A more subtle problem arises because the IA32 does not impose any data and code separation requirement, and some libraries still use dispatch tables stored directly in the code. In those cases the addresses in one of these internal tables are scrambled at load time (because they are in a code section), but are not descrambled at execution time because they are read as data. Although this does not cause an illegal operation, it causes the emulated code to jump to a random address and fail inappropriately. We solved this problem by adding machine code to check for internal references in the block written to the cache. If the reference was internal, we performed an additional descrambling operation on the address recovered as data.

An additional difficulty was discovered with Valgrind itself. For somewhat subtle reasons involving dynamic libraries, Valgrind has to emulate itself at certain moments, and it has a special workaround in its code to execute certain functions natively, avoiding an infinite emulation regress. We handled this by detecting Valgrind's own address ranges and treating them as special cases. We believe this issue is specific to Valgrind, and we expect not to have it in other emulators.

3. EXPERIMENTAL RESULTS

The results reported in this section were obtained using the RISE prototype, available under the GPL from http://cs.unm.edu/~immsec. We have tested RISE's ability to run programs successfully under normal conditions and its ability to disrupt a variety of machine code injection attacks (Section 3.1). In addition, we have tested the safety of executing instruction sequences after they have been randomized (Section 3.2) and concluded that programs randomized under RISE can execute with very low probability of doing damage. Finally, we make some observations about the performance of RISE (Section 3.3), concluding that the approach could be used in a production system if ported to a more efficient emulator.

3.1 Attacks

We tested two synthetic and a dozen real attacks. The synthetic attacks, published in [23], create a vulnerable buffer—in one case on the heap and in the other case on the stack—and inject shellcode into it. Without RISE, both attacks successfully spawned a shell, and with RISE, the attacks were stopped. The real attacks were launched from the CORE Impact attack toolkit [1]. We selected twelve attacks that satisfied the following requirements of our threat model and the chosen emulation tool: the attack is launched from a remote site; the attack injects binary code at some point in the execution; the attack succeeds on a Linux OS. Valgrind is specifically designed to run under Linux, and we tested several different Linux distributions, reporting data from two (RedHat from 6.2 to 7.3 and Mandrake 7.2).

All of the attacks were tested to make sure they were successful in the vulnerable application before retesting with RISE. The attacks were all successfully defeated by RISE (column 4 of Table 1). When we analyzed the logs generated by RISE, however, we discovered that 9 of the 14 tested attacks failed without ever executing the injected attack code (column 3). This class of attacks is notoriously fragile, and the mere fact of emulation can often disrupt them; one could imagine modifying the attacks to overcome the perturbations of the emulator, and in the future we hope to test these modified attacks against RISE.

The synthetic attacks and the more robust real attacks (Bind NXT, Samba trans2, and rpc.statd), were unaffected by the emulator's presence and all managed to establish a shell successfully when

Attack	Linux Distribution	Stopped by unmodified Valgrind	Stopped by RISE
Synthetic heap overflow	N/A		√
Synthetic stack overflow	N/A		√
Apache OpenSSL SSLv2	RedHat 7.0 and 7.2	√	√
Apache mod php	RedHat 7.2	√	√
Bind NXT	RedHat 6.2		√
Bind TSIG	RedHat 6.2	√	√
CVS pserver double free	RedHat 7.3	√	√
SAMBA nttrans	RedHat 7.2	√	√
SAMBA trans2	RedHat 7.2		√
SSH integer overflow	Mandrake 7.2	√	√
rpc.statd format string	RedHat 6.2		√
sendmail crackaddr buffer overflow	RedHat 7.3	√	√
wuftpd format string	RedHat 6.2 to 7.3	√	√
wuftpd glob " {"	RedHat 6.2	√	√

Table 1: Results of attacks executed under Valgrind (without RISE) and RISE.

the target program was run on an unmodified version of Valgrind. However, all of them were stopped by RISE. Bind NXT and Samba trans2 attacks are both based on stack overflows, while the rpc.statd attack injects binary code into the GOT table.

These results confirm that we successfully implemented RISE and that a randomized instruction set prevents injected machine code from executing without the need for any knowledge about how or where the code was inserted in process space.

3.2 How safe is it to execute random instructions?

Defenses such as RISE depend on randomization to prevent an attacker from knowing precisely what an attack will do. If foreign machine code is injected into a RISE protected program without scrambling, then when it is unscrambled for execution it will be mapped to essentially random bytes and will not perform any specific function.

If such random code does not behave as intended, what does it do? The expectation is that random code strings will cause the attacked program to crash quickly, but we don't know a priori what will happen. The RISE prototype produces randomized instruction sets that are in byte-for-byte correspondence with actual x86 instructions, so the transformation process does not affect code size or layout. This avoids complexity and allows us to defer randomization until load time. But, with so much of the x86 opcode space already defined, there is a significant chance that a randomly scrambled opcode will be something other than an illegal instruction.

To test the safety of random instructions, we performed the following test: We built a small program that contained a rootshell exploit coded in x86 machine code (the shellcode from '`test-sc2.c`' in [6]). When the program ran, it first randomized the exploit code in place using a random number seed supplied on the command line. It then 'returned into' the randomized attack code following the pattern that could happen in an attack. We ran 30,000 tests varying only the random seed, running the program under gdb to capture information about if, where, and why the program dies. Table 2 and Figure 1 summarize the results. Over 99.8% of randomizations lead to the program aborting by one of four signals. SIGILL is an illegal instruction, SIGFPE is a floating point exception (such as division by zero), and SIGSEGV and SIGBUS are two varieties of addressing problems. In the remaining cases, the program entered an (apparently) infinite loop. In none of the 30,000 test cases did the attack code manage to access the command interpreter `/bin/sh` as intended by the attacker.

Outcome		*Count*	*Percent*	*Cumulative*
Signalled		29,945	99.82%	99.82%
SIGSEGV	25,162			
SIGILL	4,504			
SIGFPE	178			
SIGBUS	101			
Looped(timeout)		55	0.18%	100.0%
Acquired shell		0	0.0%	0.0%
Total tests		30000		

Table 2: Outcomes of executing randomized shell acquisition code.

There are caveats to this data. Note that SIGSEGVs are by far the most commonly emitted signal—but that could be misleading because the test program is so small. A larger program with a correspondingly larger space of legal addresses would be expected to generate fewer SEGV's and more jumps to random but legal addresses, causing more complex and possibly subtly harmful behavior patterns.

Nonetheless, this case study suggests that the vast majority of randomizations of a genuine attack do indeed simply cause a program crash. Although this test does not directly answer the question of *how fast* the crashes tend to occur, Figure 1 provides indirect data on that point, illustrating where the program counter was when the signal occurred. There is a strong peak at 0—in over one quarter of all test cases, when the program was stopped it was on the first byte of the randomized attack code, and the fraction of attacks falls off rapidly at increasing offsets.

Another caveat in this test is that we don't know exactly how many instructions were executed before the signal occurred. Random control transfers occur frequently, so the location of a signal does not correlate directly with number of instructions executed. As seen in Figure 1, for example, a cumulative total of about 6% of the signaling cases occurred at addresses below the starting point of the attack.

Using RISE itself, we can address the question of how many instructions are executed, because it is easy for an emulator to count how many instructions it has emulated. However, it is much more expensive to collect data this way. Table 3 gives results for a few concrete data points. We show data on three real attacks against vulnerable programs, with an average of under five instructions being executed before the attacked program is stopped (column 4). Column 3 indicates how many attack instances we ran (each with a different random seed for RISE) to compute the average. As column 5 shows, most attack instances were stopped by an attempt to execute a non-existent opcode. In addition, we ran the two synthetic attacks (described earlier) one hundred times each (with a new seed each time) and discovered that neither attack ever executed successfully. On average, each synthetic attack instance executed 2.35 bytes of instructions before process death.

Within the RISE approach, one could avoid the problem of accidentally viable code by mapping to a larger instruction set. The size could be tuned to reflect the desired percentage of incorrect unscramblings that will likely lead immediately to an illegal instruction.

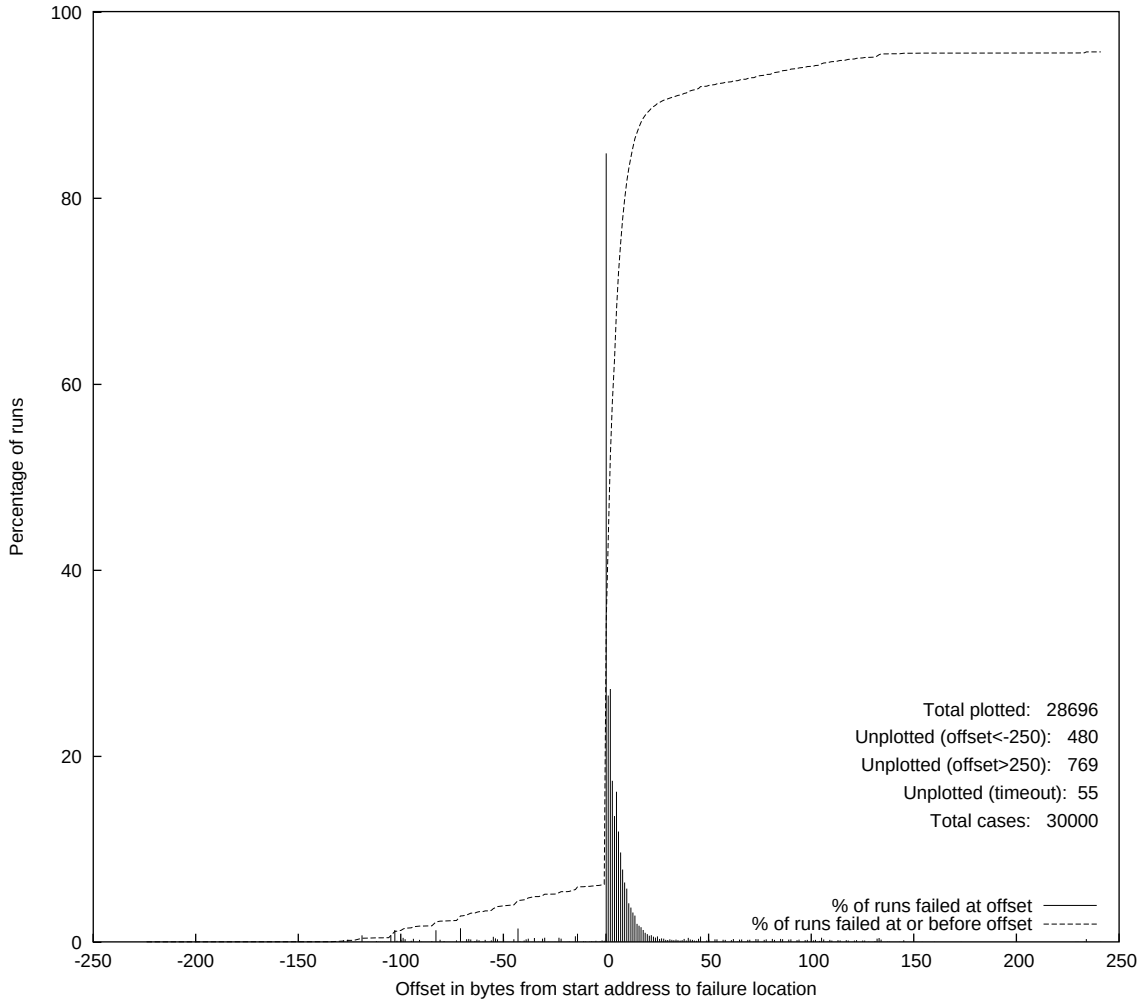

Figure 1: Distribution of locations at which the synthetic attack was stopped.

Attack Name	Application	No. of attacks	Avg. no. of insns.	Illegal insn.
Named NXT Resource Record Overflow	Bind 8.2.1-7	33	2.84	84%
rpc.statd format string	nfs-utils 0.1.6-2	25	4.13	80%
Samba trans2 exploit	smbd 2.2.1a	81	3.13	73%

Table 3: Survival time in executed instructions for attack code in real applications running under RISE. Column 4 gives the average number of instructions executed before failure, and column 5 summarizes the percentage of runs failing because of illegal instructions.

3.3 Performance

There is a significant cost introduced by the memory checking engine of Valgrind. However, RISE adds only a modest performance penalty beyond that. In terms of execution time, a RISE-protected program executes about 5% more slowly than the same program running under Valgrind; we believe much of that slowdown is due to the relatively high cost of the `mprotect` system calls used to control modifications of the trace cache. In terms of space, significant impacts come from the scrambling information and the private copies of shared libraries, each of which requires about as much space as the protected code.

We have been able to RISE-protect every one of the services used in the experiments (httpd, named, cvs pserver, smbd, sshd, rpc.statd, sendmail, wuftpd) on a 200 MHz Pentium computer with 128 MB RAM, and run it with reasonable response time. This is a far smaller and slower machine than any modern x86-based server system, which gives us confidence that the memory expense does not make the scheme impractical and would be a reasonable trade-off for increased security.

4. RELATED WORK

Our randomization technique is an example of *automated diversity*, an idea that has long been used in software engineering to improve fault tolerance [9, 34, 10], and more recently has been proposed as a method for improving security [17, 24, 19]. An approach similar to RISE, but focusing on a whole-system emulator, is proposed in [27]. Several other (nondiversifying) approaches have been developed for protecting against stack-smashing attacks, a method of code injection [40, 20, 21, 25]. Instruction-set randomization is also related to hardware encryption methods as protection against piracy and eavesdropping for specialized applications [12, 13, 22] and general purpose systems [29, 5].

4.1 Automated diversity

Diversity in software engineering is quite different from diversity for security. In software engineering, the basic idea is to generate multiple independent solutions to a problem (e.g., multiple versions of a software program) with the hope that they will fail independently, thus greatly improving the chances that some solution out of the collection will perform correctly in every circumstance. The different solutions may or may not be produced manually, and the number of solutions is typically quite small, around ten.

Diversity in security is introduced for a different reason. Here, the goal is to reduce the risk of widely replicated attacks, by forcing the attacker to redesign the attack each time it is applied. For example, in the case of a buffer overflow attack, the goal is to force the attacker to rewrite the attack code for each new computer that is attacked. Here the number of different diverse solutions is very high, potentially equal to the total number of program copies for any given program. Manual methods are infeasible here, and the diversity must be produced auomatically.

Cowan et al. introduced a classification of diversity methods applied to security (called 'security adaptations') which classifies adaptations based on what is being adapted, either the interface or the implementation [19]. Interface adaptations modify code layout or access controls to interfaces, without changing the underlying implementation to which the interface gives access. Implementation adaptations, on the other hand, do modify the underlying implementation of some portion of the system to make it resistant to attacks. RISE can be viewed as an interface randomization at the machine code level.

Earlier work in automated diversity for security has experimented with diversifying data layouts [17, 33], file systems [19], and system-call interfaces [16]. In addition, several projects address the code-injection threat model directly, and we describe those projects briefly.

In 1997, Forrest et al. presented a general view of the possibilities of diversity for security [24], introducing the idea of deliberately diversifying data layouts as well as code, and demonstrated an example of diversification that randomly padded stack frames so that exact return address locations would be less predictable, making it harder for an attacker to locate the return address and other key stack offsets. Developers of buffer overflow attacks have developed a variety of workarounds—such as 'ramps' and 'landing zones' of no-ops and multiple return addresses—aimed at coping with variations across different versions or different compilations of the vulnerable software. Deliberate diversification via random stack padding coerces an attacker to use such generalization techniques; it also necessitates larger attack codes in proportion to the size range of random padding employed.

The StackGuard system [20] provides a counter-defense against landing zones and similar attack techniques by interposing a hard-to-guess 'canary word' before the return address, the value of which is checked before the function returns. An attempt to overwrite the return address via linear stack smashing will almost surely change the canary value and thus be detected.

4.2 Enforcing security with optimizing interpreters

It has been noted that the current trend in binary-to-binary optimizing interpreters could be used for more detailed inspection of executing code, because every control transfer is detected during the interpretation process. Kiriansky et al. [28] proposed a method called 'code shepherding' in which various policies are defined to govern allowable control transfers. Two of those types of policies are relevant to the RISE approach.

Code origins policies grant differential access based on the source of the code. When it is possible to establish if the instruction to be executed came from a disk binary (modified or unmodified) or from dynamically generated code (original or modified after generation), policy decisions can be made based on that origin information. In our model, we are implicitly implementing a code-origin policy, in that only unmodified code from disk is allowed to execute. An advantage of the RISE approach is that the origin check cannot be avoided—only properly sourced code is mapped into the private instruction set so it executes successfully. Currently, the only exception we have to the disk-origin code policy is the code deposited in the stack by signals, which is handled specially by Valgrind. Also relevant are *restricted control transfers* in which a transfer is allowed or disallowed according to its source, destination, and type. Although we use a restricted version of this policy to allow signal

code on the stack, in other cases we rely on the RISE 'language barrier' to ensure that injected code will fail.

4.3 Other defenses against buffer overflows

In addition to the stack-frame padding and canary methods [40, 20] described earlier, several other solutions have been proposed to deal specifically with buffer overflows [21]. These solutions employ compiler extensions [20, 24], hardware characteristics [25, 32], kernel modifications [37, 32], library modifications [3], or static analysis [41] to prevent and detect exploitation of buffer overflow vulnerabilities.

RISE shares many of the advantages of non-executable stack and heap techniques, including the ability to randomize ordinary executable files and no special compilation requirements however. Our approach differs, however, from non-executable stacks and heaps in important ways. First, most non-executable stack/heap systems (such as PaX [32]) are applied systemwide, while RISE can be selectively employed on a per-process basis. This distinction becomes important, for example, when we consider Java Virtual Machines, where a runtime compilation process generates code, places it on the heap, then later jumps to it. In a system with a traditional non-executable heap, JVMs cannot run at all. In RISE, the JVM process can simply be run outside of RISE without compromising the security of other running processes. Second, enabling non-executable stack/heap protection on a system often requires additional hardware or operating system modification. RISE runs as a user-level application and requires no special hardware or OS changes. RISE is capable of running on any binary-to-binary translator, and so can run on any system with such software.

4.4 Hardware encryption

Because RISE uses runtime code scrambling to improve security, it resembles some hardware-based code encryption schemes. Hardware components to allow decryption of code and/or data on-the-fly have been proposed since the late 70's [12, 13] and implemented as microcontrollers for custom systems (for example the DS5002FP microcontroller [22]). The two main objectives of these cryptoprocessors are to protect code from piracy and data from in-chip eavesdropping. An early proposal for the use of hardware encryption in general purpose systems was presented by Kuhn for a very high threat level where the encryption and decryption was performed at the level of cache lines [29]. This proposal still adheres to the model of protecting licensed software from users, and not users from intruders, so there is no analysis of how to deal with shared libraries or how to encrypt (if desired) existing open applications. A more extensive proposal was included as part of TCPA/TCG [5]. Although the published TCPA/TCG specifications provide for encrypted code in memory, which is decrypted on the fly, TCPA/TCG is designed as a much larger authentication and verification scheme and has raised controversies about Digital Rights Management (DRM) and end-users losing control of their systems ([7],[8]). RISE contains none of the machinery found in TCPA/TCG for supporting DRM. On the contrary, RISE is designed to maintain control locally to protect the user from injected code.

5. DISCUSSION AND CONCLUSIONS

In this paper we introduced the concept of a randomized instruction set emulator as a defense against binary code injection attacks. We demonstrated the feasibility and utility of this concept with a proof-of-concept implementation based on Valgrind. Our implementation successfully scrambles binary code at load time, unscrambles it instruction-by-instruction during instruction fetch, and executes the unscrambled code correctly. The implementation was successfully tested on several code-injection attacks, some real and some synthesized to exhibit common injection techniques.

Although Valgrind has some limitations, discussed in Section 2, we are optimistic that improved designs and implementations of "randomized machines" would vastly increase performance and reduce resource requirements, potentially expanding the range of attacks the approach can mitigate. In the current implementation, aside from performance issues, there is a potential concern about the dense packing of legal x86 instructions in the space of all possible byte patterns. A random scrambling of bits is likely to produce a different legal instruction. Doubling the size of the instruction encoding would enormously reduce the risk of a processor successfully executing a long enough sequence of undescrambled instructions to do damage. Although our preliminary analysis shows that this risk is low even with the current implementation, we believe that emerging 'soft-hardware' architectures such as Crusoe will make it possible to reduce the risk even further.

A valid concern when evaluating RISE's security is its susceptibility to key discovery, as an attacker with the appropriate scrambling information could inject scrambled code which will be accepted by the emulator. We believe that RISE is highly resistant to this class of attacks.

RISE is resilient against brute force attacks because the attacker's work is exponential in the shortest code sequence that will make an externally detectable difference if it is unscrambled properly. We can be optimistic because most x86 attack codes are at least dozens of bytes long, but if a software flaw existed that was exploitable with, say, a single one-byte opcode, then RISE would be vulnerable, although the process of guessing even a one-byte representation would cause system crashes easily detectable by an administrator.

An alternative path for an attacker is to try to dump arbitrary address ranges of the process into the network, and recover the key from the downloaded information. The download could be part of the key itself (stored in the process address space), scrambled code, or unscrambled data. Unscrambled data does not give the attacker any information about the key. Even if the attacker obtains scrambled code or pieces of the key (they are equivalent because we can assume that the attacker has knowledge of the program binary), using the stolen key piece might not be feasible. If the key is created eagerly, with a key for every possible address in the program, past or future, then the attacker would still need to know where the attack code is going to be written in process space to be able to use that information. However, in our implementation, where keys are created lazily for code loaded from disk, the key for the addresses targeted by the attack might not exist, and therefore might not be discoverable. The keys that do exist are for addresses that are usually not used in code injection attacks because they are write protected. In summary, it would be extremely difficult to discover or use a particular encoding during the lifetime of a process.

An attraction of RISE, compared to an approach such as code shepherding, is that injected code is stopped by an inherent property of the system, without requiring any explicit or manually defined checks before execution. Although divorcing policy from mechanism (as in code shepherding) is a valid design principle in general, it is very easy to make mistakes in defining security policies, and a mechanism that inherently enforces a correct policy is desirable.

An essential requirement for using RISE for improving security is that the distinction between code and data must be carefully maintained. The discovery that code and data can be systematically interchanged was a key advance in early computer design, and that dual interpretation of bits as both numbers and commands is inher-

ent to programmable computing. However, all that flexibility and power turn into security risks if we cannot control how and when data become interpreted as code. Code injection attacks provide a compelling example, as the easiest way to inject code into a binary is by disguising it as data, e.g., as arguments to functions in a victim program.

Fortunately, code and data are typically used in very different ways, so advances in computer architecture intended solely to improve performance, such as separate instruction caches and data caches, also have helped enforce good hygiene in distinguishing machine code from data, helping make the RISE approach feasible. At the same time, of course, the rise of mobile code, such as Javascript in web pages and macros embedded in word processing documents, tends to blur the code/data distinction and create new risks.

Although our paper illustrates the idea of randomizing instruction sets at the machine code level, the basic concept could be applied wherever it is possible to (1) distinguish code from data, (2) identify all sources of trusted code, and (3) introduce hidden diversity into all and only the trusted code. A RISE for protecting `printf` format strings, for example, might rely on compile-time detection of legitimate format strings, which might either be randomized upon detection, or flagged by the compiler for randomization sometime closer to runtime. Certainly, it is essential that a running program interact with external information, at some point, or no externally useful computation can be performed. However, as the recent SQL attacks illustrate [26], it is increasingly dangerous to express running programs in externally known languages. Randomized instruction set emulators are one step towards reducing that risk.

As the complexity of systems grows, and 100% provable overall system security seems an ever more distant goal, the principle of diversity suggests that having a variety of defensive techniques based on different mechanisms with different properties stands to provide increased robustness, even if the techniques address partially or completely overlapping threats. Exploiting the idea that it's hard to get much done when you don't know the language, RISE is another technique in the defender's arsenal against binary code injection attacks.

Acknowledgments

The authors gratefully acknowledge the partial support of the National Science Foundation (grants ANIR-9986555, CCR-0219587, CCR-0085792, CCR-0311686, EIA-0218262, and EIA-0238027), the Office of Naval Research (grant N00014-99-1-0417), Defense Advanced Research Projects Agency (grants AGR F30602-00-2-0584 and F30602-02-1-0146), Sandia National Laboratories, Hewlett-Packard, Microsoft Research, Intel Corporation, and the Santa Fe Institute.

6. REFERENCES

[1] CORE Security Technologies. In *http://www1.corest.com/home/home.php*.

[2] CVS Directory Request Double Free Heap Corruption Vulnerability. In *http://www.securityfocus.com/bid/6650*.

[3] libsafe - Detect and handle buffer overflow attacks. In *http://www.gnu.org/directory/ security/net/libsafe.html*.

[4] MySQL COM_CHANGE_USER Password Length Account Compromise Vulnerability. In *http://www.securityfocus.com/bid/6373*.

[5] TCPA Trusted Computing Platform Alliance. In *http://www.trustedcomputing.org/home*.

[6] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49(7), Nov. 1996.

[7] R. Anderson. 'Trusted Computing' and competition policy - issues for computing professionals. *Upgrade*, IV(3):35–41, June 2003.

[8] W. A. Arbaugh. Improving the TCPA specification. *IEEE Computer*, 35(8):77–79, August 2002.

[9] A. Avizienis. The Methodology of N-Version Programming. In M. Lyu, editor, *Software Fault Tolerance*, pages 23–46. John Wiley & Sons Ltd., 1995.

[10] A. Avizienis and L. Chen. On the implementation of N-Version programming for software fault tolerance during execution. In *Proceedings of IEEE COMPSAC 77*, pages 149–155, Nov. 1977.

[11] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 1–12, Vancouver, British Columbia, Canada, 2000. ACM Press.

[12] R. M. Best. Microprocessor for executing enciphered programs, U.S. Patent No. 4 168 396, September 18 1979.

[13] R. M. Best. Preventing software piracy with crypto-microprocessors. In *Proceedings of the IEEE Spring COMPCON '80*, pages 466–469, San Francisco, California, Feb. 1980.

[14] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An approach to combat buffer overflows, format-string attacks and more. In *12th Usenix Security Symposium*, Aug. 2003.

[15] D. Bruening, S. Amarasinghe, and E. Duesterwald. Design and implementation of a dynamic optimization framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Dec. 2001.

[16] M. Chew and D. Song. Mitigating Buffer Overflows by Operating System Randomization. Technical Report CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University, Dec. 2002.

[17] F. Cohen. Operating System Protection through Program Evolution. *Computers and Security*, 12(6):565–584, Oct. 1993.

[18] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. Format guard: Automatic protection from `printf` format string vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, Washington DC, August 2001.

[19] C. Cowan, H. Hinton, C. Pu, and J. Walpole. A Cracker Patch Choice: An Analysis of Post Hoc Security Techniques. In *National Information Systems Security Conference (NISSC)*, Baltimore MD, October 16-19 2000.

[20] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Automatic Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, Jan. 1998.

[21] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 119–129, Jan. 2000.

[22] Dallas Semiconductor. DS5002FP secure microprocessor chip. http://pdfserv.maxim-ic.com/en/ds/DS5002FP.pdf.

[23] P. Fayolle and V. Glaume. A buffer overflow study, attacks & defenses. In *http://www.enseirb.fr/~glaume/indexen.html*.

[24] S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[25] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*, Washington D.C., August 2001.

[26] M. Harper. SQL injection attacks - are you safe? In *Sitepoint, http://www.sitepoint.com/article/794*, June 17 2002.

[27] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *www.cs.columbia.edu/˜gskc/publications/isaRandomization.pdf*.

[28] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Sheperding. In *Proceeding of the 11th USENIX Security Symposium*, San Francisco, California, August 2002.

[29] M. Kuhn. The TrustNo 1 cryptoprocessor concept. Technical Report CS555 Report, Purdue University, April 04 1997.

[30] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 58(4), Dec. 2001.

[31] T. Newsham. Format string attacks. In *http://www.securityfocus.com/archive/1/81565*, September 9 2000.

[32] PaX team. Non executable data pages. In *http://pageexec.virtualave.net/ pageexec.txt*, 2002.

[33] C. Pu, A. Black, C. Cowan, and J. Walpole. A specialization toolkit to increase the diversity of operating systems. In *Proceedings of the 1996 ICMAS Workshop on Immunity-Based Systems*, Nara, Japan, December 1996.

[34] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions in Software Engineering*, 1(2):220–232, 1975.

[35] B. Schneier. *Applied Cryptography*. John Wiley & Sons, 1996.

[36] J. Seward. Valgrind, an open-source memory debugger for x86-GNU/Linux. In *http://developer.kde.org/˜sewardj/*, 2002.

[37] Solar Designer. Non-executable user stack. In *http://www.openwall.com/linux*.

[38] Tool Interface Standards Committee. *Executable and Linking Format (ELF)*, May 1995.

[39] T. Tso. random.c A strong random number generator. In *http://www.linuxsecurity.com/feature_stories/random.c*.

[40] Vendicator. StackShield: A stack smashing technique protection tool for Linux. In *http://angelfire.com/sk/stackshield*.

[41] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

Appendix B

DESIGN AND IMPLEMENTATION OF SIND, A DYNAMIC BINARY TRANSLATOR

by

TREK PALMER

B.S. Computer Science, University of New Mexico, 2001

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Science

The University of New Mexico
Albuquerque, New Mexico

December, 2003

DESIGN AND IMPLEMENTATION OF SIND, A DYNAMIC BINARY TRANSLATOR

by

TREK PALMER

B.S. Computer Science, University of New Mexico, 2001

M.S., Computer Science, University of New Mexico, 2003

Abstract

Recent work with dynamic optimization in platform independent, virtual machine based languages such as Java has sparked interest in the possibility of applying similar techniques to arbitrary compiled binary programs. Systems such as Dynamo, DAISY, and FX!32 exploit dynamic optimization techniques to improve performance of native or foreign architecture binaries. However, research in this area is complicated by the lack of openly licensed, freely available, and platform-independent experimental frameworks. SIND aims to fill this void by providing an easily-extensible and flexible framework for research and development of applications and techniques of binary translation. Current research focuses are dynamic optimization of running binaries and dynamic security augmentation and integrity assurance.

Chapter 1

Introduction

Program transformation and optimization are not new ideas; however the notion of performing them at runtime is a recent invention. The attraction of dynamic transformation and analysis of programs derives, in part, from the fact that the amount of static information available to a compiler is shrinking. Object-oriented languages that support dynamic loading and unloading of code impose a serious restriction on the ability of static analysis to effectively guide optimization. Even in a fairly static O-O language such as C++, there are considerably more challenges for a static compiler to overcome than in C. Static compilation techniques fail primarily because they have insufficient information to work with. When statically compiling a program with loadable modules, for instance, the compiler cannot make many assumptions about the internal structure of those modules, and consequently standard optimization techniques (such as inlining) become impossible. Also, for static analysis to be fully effective, it is often necessary to have access to the source code. For instance, some of the most effective security analysis programs (such as StackGaurd [4]) need full access to the source code to correctly protect vulnerable code segments. In reality, however, source code is often impossible to obtain.

1.1 Dynamic Binary Translation

Dynamic binary translation is a method to overcome these deficiencies in static compilation techniques. The basic idea is to dynamically monitor a running program and use the gathered profile (which contains a great deal of information) to guide further program transformations. The challenge is to do this efficiently and transparently. Both efficiency and transparency are difficult problems. To provide transparency, the binary translator must emulate all the idiosyncrasies of the underlying system, and do so in such a way that control never leaves the translator. This is the problem that most debuggers have to solve. However, whereas a debugger developer can choose to sacrifice speed for convenience and reliability, such trade-offs cannot be made with a dynamic binary translator. The efficiency constraint means that the system as a whole must be able to simulate the underlying architecture without significantly slowing down the running program. In practice, this means about a 10% slowdown is acceptable. This leads to the use of many arcane tricks and techniques to achieve the seemingly impossible goal of dynamic profiling with almost no slowdown. These are what makes designing and implementing a dynamic binary translator such an engineering challenge.

Most binary translators have the same basic components: something to profile the running code and gather traces, something to transform the traces into fragments, and something to link the fragments up with the program's address space and run the fragments directly on the processor. It is the last step that makes up for the cost of profiling and transforming running code. Because most of a program's execution is confined to a small portion of the code, if that portion is optimized and run directly on the processor it would speed up execution significantly. Identifying those hot sections of the code, however, is effectively impossible to do statically. Fortunately, the runtime characteristics of many programs are often much simpler than the static whole-program characteristics. This simplicity, along with the increased information available at runtime, makes it possible for dynamic translators to identify important segments of code that static analysis

would not be able to catch. This ability can be used to guide instance-tailored optimizations to improve program performance, or to guide runtime security transformations to improve program stability and security.

1.2 Why Another One?

While SIND is not the first attempt at a dynamic binary translator, it is the first open one. Many previous systems were created by companies as either an internal research tool[1] or as a commercial product [3], and consequently were never released. As the discussion in Chapter 2 reveals, many of these systems also have deficiencies or peculiar requirements. Some systems are so tied to the target platform that porting them was infeasible [1]. Other systems have specific (and unobtainable) hardware requirements [5]. SIND was designed with all this in mind. The SIND framework was intentionally constructed to be portable and the current implementation for the UltraSPARC can run on commodity hardware with no custom components. These advantages alone warrant the development of SIND. In addition, SIND is an open-source system and as such may become an important research tool with a large developer base. It was, in fact, the very lack of such an open tool that motivated the development of SIND.

1.3 SIND

SIND is my effort at designing a platform-independent dynamic binary translation framework, and implementing that framework for the UltraSPARC architecture (running Solaris/SPARC). This effort stems from the fact that there are no real open platforms for doing dynamic translation. This hinders research, especially in a field where open research tools are the norm. SIND's design was abstracted from several published dynamic binary translators and is aimed at providing a general framework for building a binary

translator for any given platform. The whole system is organized in an O-O fashion, with each major component as a separate module. This modularity is intended to aid initial and subsequent development. In the future, it should be possible to extend a module without having to modify any other code.

The current SIND system implements user-level integer instructions. No supervisor or floating point code is currently supported. The lack of supervisor code is not a problem because in a modern UNIX-like system (such as Solaris), all the supervisor code lives in kernel space. Requests to this code are made through syscalls, which are proxied by SIND. Floating-point code is less common than integer code, but for SIND to be truly useful it must handle floating point operations. These instructions were intentionally passed over due to the potential complexity of implementing them correctly (and the resultant debugging nightmare). SIND is also unaware of the Solaris threading infrastructure and may therefore be insufficiently thread-safe.

1.4 Overview of the Thesis

The remainder of this document is organized as follows: Chapter 2 gives a summary of the previous efforts at dynamic binary translation; Chapter 3 describes, in detail, the design of the SIND system; Chapter 4 is a discussion of the current implementation of SIND for the UltraSPARC architecture; Chapter 5 is a discussion and evaluation of the performance and flexibility of the current experimental system; Chapter 6 is an overview of the history of the project as well as a discussion of large scale technical issues encountered while implementing SIND; Chapter 7 describes how to use the developmental SIND tool; and lastly the document is concluded with an appendix describing the code itself and details such as directory structure.

Chapter 2

Previous Efforts

There have been several notable efforts at dynamic binary translation. The most well-known is the Dynamo project from HPLabs. This was a dynamic optimization research project for HPPA systems running HP-UX. Another interesting project was the FX!32 project from DEC (now part of HP/Compaq). This system was a dynamic binary translator that ran IA32 binaries on an Alpha (running Windows NT). FX!32 had several notable features: not only did it efficiently transform foreign binary instructions, it persistently stored the fragments on disk and optimized them in an offline batch-processing phase. Other interesting systems include the DAISY project from IBM, the Hotspot and Jalapeño/Jikes JVMs, and Transmeta's Crusoe 'code-morphing' technology.

2.1 Dynamo

The Dynamo system [1] is, in many ways, the seminal effort in this field. It is the most popularized effort that actually achieves noticeable improvements in running time. The Dynamo system is geared toward dynamic runtime optimization of HPPA binaries running under a custom version of HP-UX. The system is bootstrapped by a hacked version of

`crt.o` and begins running the binary immediately. The instructions are fully interpreted by a software interpreter, whose primary task is to identify and capture hot code traces from the running program. The profiling method used is one of the Dynamo team's fundamental contributions. They experimented with several profiling metrics and found that a simple statistical approach yielded the best combination of accuracy and speed. First, the profiler only focuses on code traces that started with *trace heads*, namely backwards-taken branches. These branches are indicative of loops within the program, and Dynamo assumes that this is where most of a program's work gets done. Secondly, the profiler assumes that, on average, the branches being taken when it examines the code would be the ones the program would normally take. Therefore when a trace head became hot (was visited enough times), only a single code trace would be gathered.

This code trace is then run through several simple compiler passes to yield an optimized fragment. Because overhead had to be small the compiler only performs simple, linear pass optimizations. The fragments are then loaded into the *fragment cache*. Dynamo's cache holds the other fundamental contribution. Rather than just linking the fragment so that it correctly accessed program data, the fragment is also potentially linked to other fragments already in the cache. This obviates the need to leave the fragment cache from one fragment merely to have to re-enter the cache to execute another fragment. This single improvement led to impressive performance increases.

Despite its many successes Dynamo has many disadvantages. First, the system is not open. This seriously hampers research, as the tool cannot be extended when necessary. Second, Dynamo is specifically tailored to the HPPA architecture and HP-UX operating system, to which I do not have access. Thirdly, Dynamo would be difficult to port, even if the source code was publicly available, the system wasn't engineered to be particularly extensible. The HP engineers who wrote Dynamo admitted that the whole system may have to be rewritten to be useful on another platform.

2.2 DynamoRIO

DynamoRIO [2] is the successor to Dynamo. It is also a closed, proprietary system, but it is designed for the Intel IA32 (x86) architecture and has versions that run under Windows and Linux. In addition to the standard problems of building a dynamic optimization system, DynamoRIO had to overcome the enormous cost of interpreting the dense and complex x86 instruction set. After several false starts, this was eventually achieved with the use of a so-called basic-block cache. This is a form of 'cut & paste' interpretation in which the interpreter/decoder fetches basic blocks from memory, rewrites branch/jump targets and executes the modified code directly on the processor. This alleviated the difficulty of actually interpreting x86 instructions, but made profiling more complex. The initial decision to use a basic-block cache also tied the rest of the system to the x86 architecture [1]. In the cause of efficiency, each component of DynamoRIO was written to be as specific to the x86 architecture as possible. As a consequence, the entire system is highly non-portable and would have to be completely rewritten to handle a new instruction set [Personal Discussions with Derek Bruening, the DynamoRIO Maintainer].

Despite the tight coupling between DynamoRIO and the x86 architecture, the system is more open and flexible than the original Dynamo. Even with the closed nature of the underlying source code, there is a useful API that allows outside developers to add to the system. However, such outside additions are restricted by the API and are slowed by the need to pass data through an additional interface not used by DynamoRIO internals. Despite the improvements over the original Dynamo, DynamoRIO failed to fully solve the portability and extensibility problems.

[1] The basic block cache works by rewriting branching instructions and executing the basic blocks directly on the processor. Therefore, a basic block cache is very architecture and OS specific.

2.3 FX!32

FX!32 [3] is a dynamic translation program from DEC. It is designed to translate IA32 binaries to Alpha code at runtime. The whole system runs on Windows NT for the Alpha, and existed because many NT developers were either unable or unwilling to write Alpha-friendly code. FX!32 has a number of notable features. It does a great job of translating foreign binaries, facilitated primarily by the fact that the NT API is standard across both the Alpha and IA32 platforms. This allows rapid translation of system and library calls in a 1-to-1 fashion. FX!32 also optimizes the translated traces, but in a novel way. Rather than doing optimizations at runtime the FX!32 system simply translates the trace and then saved the translated version to disk. Later on, a batch job examines the saved traces and optimizes them using potentially long-running algorithms. In practice, this means that each time a user ran an IA32 application it would be somewhat faster than the time before.

FX!32 is an interesting piece of software, but it too suffers from serious drawbacks. Primarily, it suffers from the fact that its a closed-source system. DEC (and later Compaq) sold FX!32 along with NT for Alpha, and considered releasing the code to be economically impossible. Also, FX!32 is closely tied to the NT platform, which can be difficult to develop for.

2.4 DAISY

DAISY [5] is a binary translation project from IBM that performs dynamic compilation of Power4 binaries. It is similar to Dynamo in principle, but it employs more sophisticated translation and profiling schemes. This allows DAISY to do a more sophisticated analysis than Dynamo. For instance, a limited form of control flow analysis across branches and calls is performed (to eliminate as much indirection as possible). However, this added power comes at the cost of larger runtime overhead. The DAISY project obviated this

cost by creating a custom daughter-board that would house an auxiliary processor to run the DAISY system. This secondary processor only has to run at a fraction of the speed of the main processor, and the daughter board has several megabytes of isolated memory available only to the auxiliary processor. Because of this, DAISY has automatic memory protection at no runtime cost, the additional hardware also removes the distinction between the operating system and user applications. This means that DAISY can optimize both OS code and application code (and even optimize call sequences from one through the other).

Unfortunately, the DAISY project never produced a commercially-available version of the DAISY processor in hardware. All the published results came from detailed software simulation of the proposed hardware. Even if the hardware were eventually mass-produced, it was intended for use in high-end servers, and so would probably have been very expensive.

2.5 Crusoe, JVMs, and Others

Other dynamic translation projects include the Code-morphing technology used in Transmeta's Crusoe processor [7], the HotSpot and Jalapeño/Jikes optimizing JIT JVMs, and other virtual machines that employ dynamic (otherwise known as Just In Time) compilation techniques. The main disadvantage with code-morphing is that in addition to being proprietary, it is specifically tied to the Crusoe VLIW architecture. The JVM and other language virtual machine projects, although useful from a design perspective, did not contribute much to the actual construction of SIND. This is due to the virtual machines being tailored to the needs of a specific language. This means that most language virtual machines, although closer to hardware than the uncompiled program, have features useful to the source language that are difficult to map directly to hardware.

Chapter 3

SIND Design

The SIND system design is not particularly revolutionary. It is a synthesis and extension of many dynamic translator designs. Because most dynamic binary translators have to solve similar problems, many have similar designs. This similarity is encouraging, because it means that if this structure can be expressed in code, the construction of new binary translators would be reduced to extending the base modules, rather than designing the whole system from scratch.

Figure 3.1 shows the major components of SIND. The interpreter is the module that handles the dynamic execution and profiling of the running binary. The transformers translate gathered traces into fragments. The fragment cache handles fragment linking and runs the fragment code on the processor. The memory and syscall-manager handle the system specific aspects of memory protection and operating system interaction, respectively. They are separated from the other modules to ensure as much platform independence as possible. Lastly, the bootstrapper and dispatcher initialize the other modules and handle inter-module communication.

This framework is generic enough to encompass all source and target architecture configurations, and separates the components so that they may act like ‘plug-in’ modules.

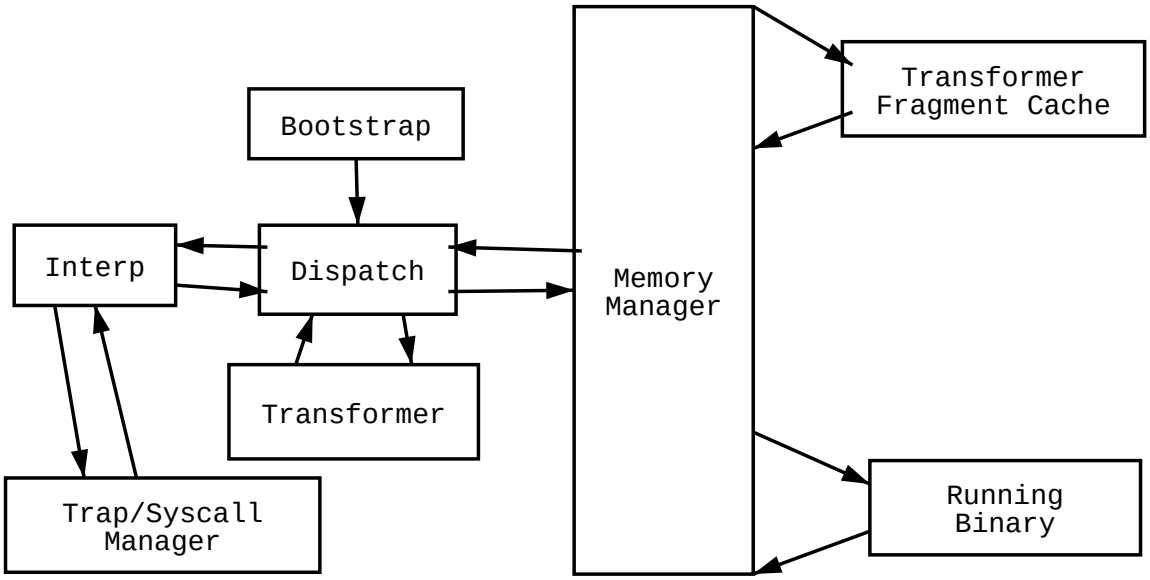

Figure 3.1: SIND modules

For instance, because the interpreter accesses memory through the Memory Manager and accesses OS functionality through the Syscall Manager, the interpreter only has to emulate the source architecture and has no dependence on the operating system. This means, ideally, that an UltraSPARC interpreter would be able to run (without modifying the interpreter source) on both an UltraSPARC and Power4 and would trust the Memory and Syscall Managers to take care of OS specifics. The intention is to isolate the interpreter from all but the most large-scale details of the target architecture. Basically, only the endianess and bit-width of the underlying system need to be taken into account (and this, only because C++ specifies no standards for the size and endianess of data).

The basic operation of the SIND framework is also platform-independent. The program to be run under dynamic translation is started by the bootstrapper. The bootstrapper assures that all dependencies (libraries and other shared objects) are loaded and halts execution just before the program starts. Then, control passes to the dispatcher, which initializes all the remaining SIND modules and starts up the interpreter. The interpreter

dynamically executes the program and gathers profiling information. According to some internal metric, the interpreter eventually decides that it has encountered an interesting code segment and gathers the relevant instructions and processor context into a trace. This trace is then handed (through dispatch) to the transformers. The transformers transform the trace into a functionally equivalent fragment. The nature of the transformations could be varied. Traces could be rewritten to be more efficient, but they could also be rewritten to be more secure, or to generate more fine-grained profiles. When the final transformer has finished its transformations, the fragment is handed to the fragment cache. The cache's primary responsibilities are to guarantee that the running fragment will have transparent access to all program data, and to simultaneously guarantee that the running fragment will not modify SIND or break out of SIND. The cache can protect SIND data by selectively write-protecting the regions of memory that SIND inhabits when the cache is entered and un-write-protecting the regions when the cache is exited. The cache also needs to check for system calls that might un-protect the SIND memory regions. It can do this by placing itself between the executing fragment and the eventual system call, and checking on the parameters passed in by the fragment. The cache can guarantee that an executing fragment will be able to access all program data by performing a final rewriting of the fragment in a process analogous to dynamic linking and loading. From then on, when program control reaches the address of a fragment, control is passed to the cache, and the fragment then runs directly on the processor. When control leaves the fragments in the cache, the SIND interpreter starts up again and continues dynamic program execution.

3.1 Interpreter

The interpreter's main function is to gather profiling information and code execution traces. These are passed to transformers, which use the profiling information to guide specialized transformations of the code traces. Because one of the goals of the SIND sys-

tem is to do runtime binary optimization, it is vital that the interpreter should introduce as little overhead as possible. As a consequence, the interpreter must be very efficient and every reasonable effort must be made to improve its speed.

The first interpreter to be fully designed and implemented in SIND emulates the 64-bit SPARC v9 architecture. The design was motivated by several factors: first, because SIND runs in non-privileged mode, the interpreter is primarily a non-privileged instruction interpreter; second, the interpreter only needs to be functionally correct, therefore no complicated hardware structure needs to be emulated in order to produce accurate simulation. The interpreter's job is then to replicate a user's view of the processor and discard any lower-level structure that interferes with the efficient execution of code.

3.1.1 Registers

The interpreter replicates user-visible registers as an array of 64-bit quantities in memory. On a 64-bit host machine these are native unsigned 64-bit integers; on 32-bit machines they are two-element `structs`. There are several caveats, however. The SPARC architecture supports register windows for integer registers. This was emulated by allocating a large array of 64-bit quantities, setting the lowest 8 to be the global registers, and having a window of 24 registers slide up and down the array as procedure calls are made and registers are saved and restored. It is important to be able to restore the user stack in order to be able fully to emulate a system call. It is also important to keep SIND's own stack separate from the user stack, because the interpreter runs in the address space of the user process and so in principle the user process's stack entries might clobber the interpreter's stack. Creating and maintaining two separate stacks is discussed below, but the discussion of restoring the user stack is in the Syscall Manager section.

Maintaining two separate execution stacks requires a bit of system hacking. The last valid stack frame is left alone, and its stack pointer (pointer to the top of the frame) is saved

for reference. A new page is allocated for the separate stack, and its topmost address is recorded. This topmost address is to become the new frame pointer. Then an explicit `save` instruction is issued; it creates a new register window, but with the stack pointer pointing into the new page. Then the frame pointer register can be manually set. From that moment on, all further calls should write their stack data to the alternate stack page(s). Apart from protecting the SIND call stack from manipulation by the interpreted program, this also means that SIND's stack can be `mprotect`-ed to safeguard its contents when executing code directly on the processor (either issuing traps or when in the fragment cache).

The floating point registers on the SPARC consist of three overlapping sets of 32, 64, and 128-bit floating point registers. There are 32 32-bit, 32 64-bit registers, and 16 128-bit registers. The 128-bit and 64-bit registers overlap completely (e.g., the first 128-bit register is the same as the first two 64-bit registers), and the 32-bit registers overlap with the bottom half of the other two. This was implemented as a contiguous region of memory, accessed in different ways depending upon the instruction used (some checking had to be done to make sure no accesses were attempted to non-existent 32-bit registers).

3.1.2 Instructions

Although the SPARC v9 architecture is 64-bit, the instructions are still 32-bit, which allows backward compatibility (consequently, the software interpreter is also capable of running SPARC v8 code). The SPARC has 30 different instruction formats, grouped together into 4 major families. However, these formats are all the same length (32 bits) and were designed to be quickly parsed by hardware. This permits streamlining the fetch and decode portions of the interpreter. Each instruction format was specified with its own bit-packed struct, and all such structs were grouped together in a union with a normal unsigned 32-bit integer. Each format family is distinguished from the others by the two high-order bits of

the instruction.[1] Thus the interpreter has jump tables for each instruction format family (actually three jump tables and one explicit function call), that are keyed by the opcode, whose position depends upon the format family. A case statement branches on the two most significant bits to the correct jump table, and then the correct function is called.

3.1.3 Exceptional Conditions

Occasionally during execution, an instruction will cause an error. The SPARC v9 architecture manual clearly defines these exceptions, and, for each instruction, specifies which exceptions it can raise. Many of the exceptions are caught by the operating system and used to handle things like page faults and memory errors. Non-recoverable exceptions usually cause the operating system to send a signal to the executing process. To mimic this, if the interpreter decides a given instruction would cause an exception (such as divide-by-zero), then a procedure similar to that used for system calls can be used. The running binary's state is restored on the stack, and then the interpreter executes the offending instruction directly on the processor. This generates the appropriate operating system action (usually, killing the process).

3.1.4 Signals and Asynchronous I/O

In the Solaris operating system there are really only two ways of communication between user and supervisor (kernel) code. One, the system call or trap, is discussed in the Syscall Manager section. The other, signals, had to be dealt with differently. Because the SIND system is guaranteed to be loaded before all other libraries, its definitions of functions will take priority (if they're exported). The SIND interpreter interposes on the signal

[1] To be precise, Format 3 and Format 4 both can have the values 10 or 11 in their upper bits, but in SIND Format 3 instructions are instructions with 10 in the upper bits and Format 4 instructions have 11 in the upper bits.

functions, and registers a special handler for all registrable signals. Thereafter, when the interpreted program registers a signal, it will go through SIND's registration system, rather than the system's. This means that SIND has to record the signal handlers registered by the program (in order to execute them when a signal is generated). When the OS sends the process a signal, it will be first intercepted by SIND, which will need to start interpreting the handler registered for that signal. In this way a signal cannot cause control to leave the SIND system.

3.2 Memory Manager

The SIND memory manager provides a generic interface between the process's address space and SIND's (possibly separate) address space. In the non-architecture specific memory manager, the interface consists of a small number of memory access and modification functions. To access memory, there are `ReadByte`, `ReadHalf`, `ReadWord`, `ReadDoubleWord`, and `ReadQuadWord`. These functions take an Address argument (the size of which is determined at compile time), and return the data located at that location. The names are meant to convey the size of value returned and follow the modern RISC convention of a word as a 32-bit value. But this does not mean that somehow the memory manager interface is only appropriate for RISC machines. To modify memory there are corresponding `Write ...` methods.

The interface exists as a separate module to support variations of SIND in which the interpreter and transformer exist in separate address spaces. In fact, the initial SIND system did exist in a separate space and accessed the processes address space using `procfs` [9]. For the current incarnation of SIND (which inhabits the process's address space), the memory manager just checks the address (to make sure that the program is not modifying SIND), and dereferences it appropriately. The `Read` and `Write` methods are also prefixed with `inline` directives to further eliminate overhead.

3.3 Syscall Manager

Because SIND sits on top of the kernel and only interprets user-space code, system calls must be executed directly on the processor to initiate the correct kernel action. SIND cannot simply execute the trap instruction directly, however the interpreted process's state must be reincarnated in the hardware, and then the trap can be issued correctly (returning into SIND, of course).

To restore the user stack, the original stack top (before control was passed to SIND) address must be preserved. The simulated stack (in the register windows array), must then be copied over to the stack area before the system call can be made. However, just copying the registers is insufficient. Each stack frame may have an arbitrarily large spill area, and that must also be preserved and copied over for trap emulation. Each time a save instruction is issued, it is remembered so that the stack offset for each frame can be properly reconstructed. However, the spilled variables do not have to be remembered. Because the interpreter is executing in the same address space as the target process, values written to the spill area will be at the correct location for the stack, so the stack frame and its corresponding registers just need to be copied around such spilled variables. However, even this is not enough to completely recreate the user stack. The register window state must also be replicated in the underlying processor. Basically, this means 'rolling back' the current stack to its state when SIND took over and then pushing on all the necessary frames. When rolling back the stack, it is necessary to save the stack frames as they are deallocated (because they will need to be restored before normal execution can resume). In practice, because the two stacks are kept separate, this means `mprotect`-ing the interpreter's stack area and issuing the necessary number of `restore` instructions. When the stack has been rolled back to its starting position, the simulated register windows need to be copied to the processor and then explicitly saved to the stack. Although this is also time-consuming, we get register saving around spilled variables automatically.

3.4 Trace Gathering

The identification and collection of traces happens within the interpreter but can be considered a separate subsystem. This is possible because the trace identification and gathering code is completely independent of the rest of the interpreter (but for efficiency reasons is part of the interpreter object). Identification of traces happens at the instruction level; an instruction with the potential to be interesting is identified as it is being emulated. On the side, the trace gathering system maintains a data structure (currently a hash table) that contains all encountered trace heads. When an interesting instruction is being emulated, the tracing code checks to see if it has been encountered before, increments its counter if it has, and inserts a new record if it hasn't. In the current experimental system, the trace gathering code looks for branches whose target is behind them (a characteristic signature of a loop). In order to avoid gathering traces for rarely executed code a certain number of iterations have to pass before a trace head can be considered hot. Currently, the system also has a threshold of 15, which means that on the 15th execution of the same potential trace head the instruction is assumed to be a genuine trace head and trace gathering can begin in earnest.

A trace is simply a sequence of instructions gathered after encountering a trace head. There are no restrictions placed on the termination conditions for a trace, however the current system will stop gathering instructions if a certain numerical limit is reached (currently 256), or if the trace head is encountered again (indicating that we have completed one iteration of the loop). Note that these restrictions do not prevent a trace from including a complete function call (and the corresponding function's body), so function inlining is essentially free. A trace is essentially an array of instructions, each of which may have some annotation (for instance, it might be useful to record whether or not a branch was taken). A trace also includes an array of registers that contain the machine state when the trace head was encountered. This information is then passed on to the Dispatcher for subsequent transformation and insertion into the fragment cache. If these operations com-

plete successfully, the entry in the trace-head data structure is updated, so that next time the interpreter will jump directly to the fragment cache, rather than interpreting the trace.

3.5 Transformers

A SIND transformer has a simple interface: it accepts a trace and returns a trace. In this way transformers can not only optimize a particular machine code trace, but can be used to convert a trace from one machine to another, or to convert a trace from machine-level to an intermediate form more appropriate for optimization. Input validation is accomplished by extending the base trace class to a concrete, platform-specific version, and writing the transformers to use the most specific trace class. Then, if an incorrect type of trace is handed to a transformer, the type error will be caught during compilation. The proper sequencing of Transformers is the job of the dispatcher and is described in detail later.

3.6 Fragment Cache

The fragment cache has a simple interface. Apart from constructors, the fragment cache has only two real methods; the first takes a new trace and inserts it into the cache, and the second executes a fragment (keyed by the program counter) stored in the cache and returns the new processor state (for the interpreter). Internally, however, these functions are far from simple. The fragment cache maintains three sets of data. The first is the fragments themselves, the second is the fragment prologues, and the third is the fragment epilogues. The prologues act as guard code to fragment entry. They perform any checks specific to a fragment. These checks may be dictated by the types of transformations applied (for instance, constant folded instructions may need to check and make sure the constant hasn't been changed). The epilogue serves as the fragment's only exit point. All possible exit points (such as branches, `call`s, and `jmp`s) have their target addresses changed so that

on exit from the fragment code, they jump to the epilogue. The epilogue's job is then to clean up after the fragment, capture the processor's state, and then jump to the fragment cache's function to return control to dispatch.

The insertion function (`newTrace()`) takes a fragment, generates a prologue and an epilogue, and rewrites crucial instructions to correctly jump to the epilogue. This function then has a reduced instruction parser that looks for these crucial instructions, rewriting them as it copies them from the trace into the cache. The execution function (`jumpToCache()`) has a simpler job: it jumps directly to the prologue code for the appropriate fragment. After the fragment has finished, `jumpToCache()` packages up all the new processor context in a class wrapper (`FragExitContext`) and returns to the dispatcher, which updates the interpreter's state accordingly.

3.7 Bootstrapper and Dispatcher

3.7.1 Bootstrapper

The SIND bootstrapper has the task of correctly halting normal execution, initializing the SIND modules, and then transferring control to the interpreter. The current incarnation of SIND for Solaris/SPARC is a preloaded library that loads itself before any other libraries (excluding ld.so). The bootstrapper runs as the `.init` function for this preloaded library. The bootstrapper first initializes the SIND modules in memory, then sets SIND's signal handler to handle all handleable signals, and finally overwrites the first two instructions from the `_start` symbol with a call to the `SINDstartup()` function. Therefore, when the bootstrapper initialization function has finished, all signals generated during normal program execution will be correctly intercepted by SIND and when all the initialization routines of all the dependencies are done, control will return to the bootstrapper.

The `SINDstartup()` function's job is to correctly capture the state of the process at the `_start` symbol and then jump into the interpreter's `executeLoop()` function, which does the actual interpretation. The process's state is captured with a bit of SPARC slight-of-hand. The address of the interpreter's registers is loaded into a register known to be zero, and then each register is stored into the interpreter's register array at the correct offset. `SINDstartup` then allocates several pages for SIND's separate stack, initiates the new stack with an explicit `save`, and overwrites the new framepointer (passed through global registers). Thus `SINDstartup` forms a buffer between the process and SIND itself, but because `SINDstartup` has no immediate variables of consequence (everything is stored in the interpreter object), the user process can overwrite its stack frame without adversely affecting SIND.

3.7.2 Dispatcher

The Dispatcher's main purpose is to serve as a common interface through which all the SIND modules can interact. In particular the dispatcher coordinates all the details of trace processing. The transformers are akin to optimizations in an optimizing compiler and it is the responsibility of the dispatcher to properly sequence the transformers (as well as handle any shared data needed by several transformers). This ordering is very specific to the transformers used, and is an additional detail not necessary to the functioning of either the interpreter or the fragment cache. Once all the transformers are done, the dispatcher then hands the annotated machine instructions to the fragment cache. If, for instance, the transformers operate on a different instruction set (either an intermediate representation or a foreign architecture), the dispatcher will need to further transform the code to the architecture that the fragment cache wants.

Chapter 4

SIND Implementation

4.1 Overview

SIND is currently implemented in a subset of C++ that is basically C, but with classes rather than structs. This exploits the convenience of C++'s object-oriented infrastructure, but avoids the more troublesome aspects of the language, such as templates and iostream. I/O is done using the traditional C functions, but dynamic memory is allocated with the `new` operator. The bulk of the SIND system is implemented as classes (the only exceptions being the bootstrapper and the signal handling code). The classes form interface-implementation pairs, where an implementation inherits from its parent implementation and implements an interface that inherits from its parent's interface. Although this is technically multiple inheritance, only one non-interface class is inherited from so all the complications endemic to C++-style multiple inheritance are avoided. The following figure 4.1 illustrates this relationship in terms of the abstract `CPU` class and the concrete `SPARCCPU` class.

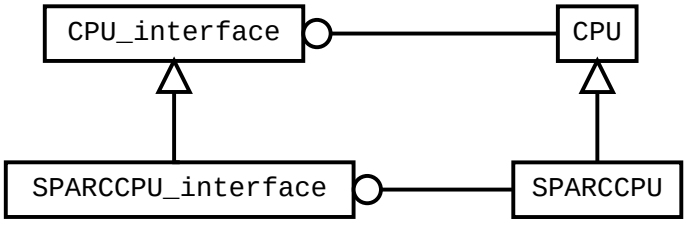

Figure 4.1: `CPU` inheritance tree

4.2 Interpreter

The SPARC interpreter is implemented in the class `SPARCCPU`. Internally, the class contains the array of registers needed to implement register windows (as described in subsection 3.1.1). `SPARCCPU` also contains representations of all user-visible registers (like the condition code registers). The only other significant amount of internal data to the interpreter are the jump tables for instruction decoding/execution. There are 4 format classes of instructions in SPARC, and each one is keyed by the 2 high order bits in an instruction word. Format 1 has only 1 instruction (the `call` instruction) and so when decoded, the method implementing `call` can be called directly. All other formats need further decoding. However, SPARC is RISC, so this decoding is very simple. Each format family has an additional op specifier (for instance Format 2 has an `op2` field), and the specific instruction is keyed by this op field. Therefore the methods implementing instructions are grouped by format into jump tables, and a switch (on the format number) will then call the function stored at the op specifiers offset in the appropriate jump table.

All of the instruction implementation methods return an `int`. Following the standard C programming style, a return value of 0 indicates success. Non-zero returns indicate failure. A positive result indicates a SPARC exception (such as *divide-by-zero*). These need to be emulated correctly (as described in subsection 3.1.3). Negative return values indicate a SIND internal error (such as an unimplemented instruction, or strange stack usage). These will usually cause the interpreter to spit out a error message and exit. The

error conditions are defined in the `SPARCExceptions.h` file.

In general, the implementation of the interpreter was a straightforward if time-consuming task. Certain instructions were non-trivial, however, and a source of constant frustration. `save` and `restore` are two particularly notable members of this category. Their implementation was complicated by all the array manipulation they had to perform. Off-by-one errors, or incorrect specification of boundary conditions were a particular problem.

4.3 Bootstrapper

The bootstrapper is perhaps one of the ugliest and strangest pieces of C code I have had to deal with. The current version is actually the third iteration of bootstrappers for SIND. It exploits the `LD_PRELOAD` functionality to load itself into the process's address space, and overwrites the first two instructions to jump to SIND. Getting this to work involved a lot of research into the Solaris linking infrastructure [8] [6] [11]. The bootstrapper has to locate the `_start` symbol, and then overwrite part of its code (which involves some mucking about with `mprotect`). The implementation of the bootstrapper was plagued by all the problems associated with system code. Poor documentation leading to poor code, strange documentation leading to strange code, and of course, the ever-present threat of undocumented features. The bootstrapper's greatest nemesis was in the form of such undocumented behavior. To locate `_start` the bootstrapper uses the `dlsym()` function to find the address. According to the documentation for the `dlfcn` functions, `dlsym()` needs a handle to the loaded object in order to work correctly. Such a handle is provided by a call to the `dlopen` function. Normally, `dlopen()` is used to open dynamic objects on the fly (as Apache does with its loadable modules), but if handed a NULL rather than a filename, `dlopen()` will return a handle to the current object. This is the method recommended in the documentation, yet when `dlsym()` was handed this handle it returned a strangely offset value of `_start` (it was, in fact, always 200 bytes away from the real

location). This odd behavior led to numerous errors, and was finally dispelled by consulting independent code examples (that is, code not written especially for the Solaris linkers guide[11]). This is a problem known to a small group of people on line, and has something to do with accessing symbols that begin with a "_". The solution is to use another form of `dlsym()`, and hand it a NULL handle. This will default to the current object.

4.4 Fragment Cache

The fragment cache was another tricky piece of coding. Although the SPARC instruction set is RISC, control transfer instructions are scattered throughout all the format families, so the mini-decoder could not be compact. As the instruction trace is stepped through, each instruction must be checked to see if it can leave the fragment. If it can, its target address (or the code following it, if not taken) must be adjusted to jump into a part of the epilogue. This part of the epilogue is then generated which will remember the PC of the exiting instruction and then transfer control to the state capture/cleanup code. Because the current fragment cache doesn't do any fancy internal linking, the code is not particularly complex, just rife with detail. Needless to say, this code is the resting place of many bit level errors and other demons of the assembly/machine code world.

Chapter 5

Evaluation of SIND

5.1 Performance of SIND

To measure the performance of SIND, several sample programs were run, and their execution times compared against those when using SIND. The programs in question were the simple test programs used to debug SIND itself. These programs were used rather than conventional benchmarks because SIND cannot currently run SPECint to completion.

5.1.1 Speed of Interpretation

The interpreter's speed was measured by timing the main loop of the `executeLoop()` function in the interpreter. The timing was performed using the high-resolution timing facilities of Solaris. Trace-gathering and fragment caching were both turned off, so the execution was completely within the interpreter. Debugging statements were also disabled[1].

As the table shows, the emulator introduces a slowdown factor of 150 to 225. This

[1]Initially, I erroneously gathered data with them enabled first, which allowed me to see that SIND's voluminous debugging output causes an additional slowdown of roughly a factor of 20.

test	description	slowdown
sind_test	arithmetic and logic test	140x
sindIO_test	hello world	210x
sindTimingTest	timing calls and I/O	223x

Table 5.1: SIND interpreter slowdown

multiple is consistent with an inspection of the compiled code for the SIND interpreter. The simple arithmetic functions have little or no branching statements and are between 100 and 150 instructions in length. The more complicated instruction functions, such as those for branching, are many thousands of instructions long, although most of that code lies in mutually exclusive branch targets. Inspecting a trace of instructions running from the initial decoding, through condition code checking, etc., shows instruction counts of between 200 and 250. Traps are the longest running instruction. The `tcc` SPARC instruction has to do all the condition code checking of a normal branch, but control is passed to the `handleTrap()` function, which must restore program state to the processor and execute the trap directly on the hardware. The situation is even worse for a syscall (`ta 8`), because it must pass through the `handleSysCall()` function first, which checks the arguments to make sure that the syscall will not side effect SIND itself. Fortunately, traps are rare.

Most of the interpreter's time was spent in the linking code. A program such as hello world, will require the execution of roughly 19,000 instructions [2]. Most of the time is spent in the linker back-patching the procedure linkage table with dynamic function addresses.

[2] Specifically, 19177 instructions.

5.1.2 Speed of Cached Fragments

Because the current system has no translators of note, the fragments in the cache are not very different from the gathered traces. The branch and jump targets have been rewritten, but no instructions have been eliminated. As a result, the fragment executes at almost the same speed as the native code. The prologue only adds 2 instructions (currently), although future transformers could enlarge this. Although the epilogue is many instructions long, it is not executed in its entirety. On an exit from a fragment, only two instructions would be executed. The first instruction is to jump to the fragment cache exit code, and the second loads the exiting PC value to a memory location (this instruction is in the branch delay slot). Therefore, the fragment is only 4 instructions longer than the original trace.

5.2 Memory Footprint of SIND

Almost all of SIND's data is statically allocated before hand or is on the stack. Those data structures that use dynamic memory have a default constructor that statically allocates a fixed amount. SIND can be built so that it allocates no dynamic memory. In this case, SIND occupies 440K of space [3]. This footprint will grow, however, as SIND is made self-contained (as explained in 7.2).

5.3 The Agility of SIND

The current experimental SIND system is capable of running 'toy' programs such as `hello world`. Current development efforts are focused on getting SIND to execute the SPECint2000 benchmarks (with the exception of the mtrt benchmark which is multi-threaded). Because the toy programs generate only a few long running loops, the fragment

[3] Actually 451568 bytes.

cache is insufficiently tested by them. Therefore, the bugs in the current system that prevent execution of SPECint are almost certainly in the fragment cache.

Chapter 6

SIND: A History

6.1 A Brief History

SIND began its life as a class project in a Java seminar (Spring 2001). Dino Dai Zovi and I were inspired by the Dynamo paper to try and implement a similar system for the SPARC. Very shortly we discovered why it took a small team of HP engineers over a year to construct Dynamo. We began with the ISEM [1] source, but soon ran into two limitations. First, ISEM only interprets 32-bit SPARC v8 code, and second (and more importantly) ISEM is a whole system emulator. ISEM emulated both privileged and non-privileged instructions, as well as a fully-featured memory subsystem and S-bus-style system bus. This was too much overhead to deal with, and after a summer of sporadic hacking, I decided that it would take less time to code an UltraSPARC emulator from scratch than to rip out the salient parts of ISEM. Also, initially we used `ptrace` [10] to access the running process, and had lifted most of our (poorly understood) bootstrap code from gdb. This led to a whole stream of difficulties with `ld.so` and memory protection. By the Christmas of 2001, I had given up on the ISEM `ptrace` combination and started coding the SPARC v9

[1] The Instructional SPARC EMulator, a 32-bit SPARC v8 emulator written by Barney Maccabe

interpreter. Dino Dai Zovi elected to start a PowerPC interpreter, but because of conflicts with work, school, and other research obligations could only make infrequent and minor contributions to SIND.

By the following summer I was the only developer working on SIND, and was mostly finished with the core interpreter. By the fall, the interpreter was sufficiently complete to warrant construction of a new bootstrapper. This lead to several branches of experimentation. Initially, I attempted to use the Solaris `procfs` facilities, but this proved to be overly complex and fraught with peril. I then decided that in the interest of both efficiency and ease of coding, I would locate SIND in the same address space as the running program. There were several implementation options: I could modify a system program (such as `ld.so` or `libcrt.o`) to automatically load SIND whenever a program was loaded; or I could use the `LD_PRELOAD` trick to cause SIND to be loaded as a shared object. The former option had the disadvantage of making SIND difficult to install and maintain (after all, I would have had to keep up with the latest release of whatever system program I modified), and the latter option had the disadvantage that my code would have to be massaged into position-independent library code. The latter option was more general and ultimately easier however, and so, compelled by my innate laziness, I made SIND a preloadable library.

The bootstrapper itself was the scene of some vicious entanglements with the Solaris Operating Environment[6]. I initially caused execution to return to SIND code by `mprotect`-ing the `_start` symbol's page to non-executable. Thus, when control finally transferred to the target program, a segfault would be triggered and intercepted by my handy bootstrapper cum signal handler. This had the disadvantage of being inexact and limited by the restrictions on actions inside signal handlers. This approach was abandoned by Spring of 2003, and replaced with the current bootstrapping system. The new system is exact and considerably more complete than that which it replaced. The spring also saw the stabilization of the interpreter, the beginnings of a trace-gathering system, and the

introduction of the Syscall Manager.

6.2 Experiences from the Design and Implementation of SIND

The design of the interpreter itself presented several challenges. There are two main options for an instruction set interpreter. It can be a full-fledged software interpreter, emulating the source instructions in software, or, if we are planning on running it on the same architecture as the instructions, we can do a 'cut-and-paste' interpreter. The cut-and-paste solution (otherwise known as a basic block cache) works by copying each instruction encountered to an area in memory, remembering to rewrite control-transfer instructions to jump to the correct new locations and then executing these copied instructions directly on the processor. This is a very lightweight interpretation system, and because it requires a decoder only capable of distinguishing control transfer instructions from the rest, it is the preferred solution on x86 platforms (systems such as Valgrind [12] and DynamoRIO [2]). However, such cut-and-paste systems have one major disadvantage. They can only be run on the platform whose instructions they are interpreting. This presented a disadvantage for our work, because we not only wanted to explore dynamic optimization, but also foreign binary execution (a lá FX!32 [3]). If we wanted to run this interpreter on another platform, it would have to be a full-fledged instruction set emulator.

The core interpreter itself is not complicated: emulating a compact RISC machine is not too difficult. Most of the effort went into the bootstrapper and system call subsystems. The bootstrapper itself has gone through many permutations. In the end, there were two major options. Either the interpreter starts itself up in the library initialization routine, or it causes control to transfer from the target binary's `_start` symbol into the interpreter. The problem with the first option is that it halts the loading process halfway through.

Normally the binary and all its dependencies are loaded and then, in the loading order, all the dependencies have their initialization routines called. If SIND were to take over in its initialization routine, then it would have to act like the loader and finish process loading. The second option, though it appeared to be more complicated, actually turned out to be the easier route. In SIND's initialization routine, the bootstrapper `mprotect`s the loaded `.text` segment to allow writes. The first two words/instructions after `_start` are saved to a reserve area, and then are overwritten with an explicit call instruction into the interpreter's code. This means that SIND will only be started *after* the loader has finished. This system is imperfect, however. If any loaded library prevents control from transferring to the start symbol (such as, for instance, by never exiting the initialization routine), then SIND will never be entered. This is not a big problem, however; because the SIND bootstrapper can easily be replaced without affecting the interpreter, a more thorough system can be developed and inserted without difficulty.

The syscall subsystem was discussed thoroughly in the design section above and took time to develop simply because of its complexity (almost all owing to the use of register windows). Development on the whole system was hampered by several tool deficiencies. The debugger we were using (gdb) has only limited support for 64-bit objects, and this severely hampered diagnosis. The debugger was also of limited use because we were not, in fact, debugging the running program: we were debugging a library that was loaded with the Unix `LD_PRELOAD` facility. Trying to use gdb's built-in facilities turned out to be more trouble than it was worth. The best method we discovered was to compile SIND with debugging on (and explicit stabs support), and cause an intentional segfault (by dereferencing `NULL`) near the suspect method. We could then load the core into gdb, and it would often give us enough information to help with debugging. When we needed more control, we inserted an `__asm__` block with an explicit debugger trap (`ta 5` on Solaris/SPARC).

Debugging, in fact, has proved to be the most complicated part of developing SIND. In an effort to make debugging easier I developed two tools. Both were actually modifications

to SIND in order to capture state correctly. The first was the driver. This was a program that stood between SIND and the running program and hand-fed SIND an instruction at a time. This was useful for debugging individual instructions in the interpreter, but not at all useful for dealing with all the strange interactions possible when SIND was in the process's address space. When debugging SIND in the process's address space, simply dereferencing NULL to cause a segfault was insufficient. `gdb` would not correctly load the preloaded library and so certain things (like `%sp` and the PLT) would be different than when the program crashed. In order to get a clear picture of memory at the point of interest, a 'BOGUS' flag was added to the SIND build process. When compiled with BOGUS, the SIND binary would insert itself into the running program's address space, and trigger a transfer of control at `_start`. However, control would not enter the interpreter, instead the overwritten instructions would be replaced and the PC of interest would be overwritten with a `jmp` to the BOGUS function that would print out the relevant parts of memory and then quit. These values would be those computed by the binary on the processor itself, and could then be compared against the voluminous debugging output of the normal (that is, not BOGUS) SIND.

The GCC compiler itself introduced problems. We were originally using the gcc 2.95 compiler collection which had somewhat buggy 64-bit support. The biggest problems were with C++ name mangling. In older versions of gcc, symbols defined in .c files or header files whose implementations were in .c files used normal C linking. That is, a function defined as `void foo()` was exported as the symbol `foo`. In gcc3 and up, anything touched by a C++ file was made to use C++ linking. C++ linking involves a technique known as *name-mangling*, whereby the symbol name has characters appended or prepended to it that the system uses to extract type information. Therefore a function `foo` in a class `bar` gets mangled to something like `_ZNKbar1fooEv`. This meant that many of the function interpositions we had created were no longer working when we upgraded to gcc 3.2 because their symbolic names were mangled beyond recognition. The way around this was to devise macros to enclose C-style definitions in a way that tells the

C++ compiler to leave them alone.

The GCC compiler also caused problems with register usage. On Solaris/SPARC systems, a shared object cannot write to the `%g2` or `%g3` registers (which are dedicated to passing values to syscalls). With gcc it is simple enough to specify not to use either global register; however, it is not possible to tell it to only avoid *writes* to those registers. This means that any code we have that explicitly copies values from `%g2` or `%g3` has to be compiled separately and then linked in later, which is cumbersome. On a load-store architecture, it should be trivial for an assembler to determine whether an expression that references a register is writing to it or just reading it!

Chapter 7

Using SIND

7.1 Invoking SIND

Because SIND is force-loaded into the running binary's address space, SIND must be a dynamic object. When SIND is finished building, there will be a file named `libsind.so` in the directory. To use this, a shell script is provided (`run_sind`). This will set `LD_PRELOAD` correctly and invoke the supplied binary. For example to run the hello_world program, one would simply type '`run_sind hello_world`'. There are some caveats, however. Because SIND uses `LD_PRELOAD`, there are some restrictions. To prevent an attacker from installing a malicious library in their home directory and causing everyone to interpose with their dangerous code, Solaris (and other ELF OSs) requires that any setuid program can only load `LD_PRELOAD`ed libraries from certain 'safe' areas. Under Solaris this can be configured using the `crle` program.

7.2 Extending SIND

The current experimental SIND system is not functional enough to be of much use to the average user, but because of its progressive open source licensing, SIND can be readily extended by needy coders. At the moment SIND will only work on 64-bit Solaris/SPARC system. An effort has been made to make the system capable of running on a 32-bit platform, but that code has not been tested and is certain to contain numerous bugs. SIND has also not been rendered self-contained, which is to say that I/O functions and memory management have not been replaced with local, specialized functions. For programs that 'play nice' this isn't a big problem. But for any advanced application (that may define its own `new` operator, for instance) SIND may fail horribly. Therefore SIND's I/O and memory needs must be met within SIND. This should not be a particularly difficult task, however. Many of SIND's data structures are fixed size and so dynamic memory is rarely used. In addition, when debugging is disabled, SIND performs no file I/O, and contains only a few print statements. The current SIND system also has no real threading or multi-process support. Although when a new process is created (with `fork()`), SIND should be copied along, this hasn't been tested.

For an overview of what functionality is in which files, please consult appendix A. That appendix also contains details on building SIND.

7.2.1 System-Dependent Code

In SIND terms, the system dependent code is that code that is either OS specific, or architecture specific. Basically, that means any code which depends upon low level system behavior, or has inlined assembly code (for state capture, for instance). In the current version of SIND, the bootstrapper and the trap handling code (`SPARCTrap.cc`) are dependent upon both the processor and operating system. The fragment cache is depen-

dent upon the processor (it processes SPARC code), and the signal handling code (signal_handling.c) needs a system with support for POSIX signals (it handles both `signal` and `sigaction`).

7.2.2 System-Independent Code

The rest of the SIND code is intended to be platform independent (insofar as C or C++ can be). The interpreter (`SPARCCPU.cc`) is meant to run on both 64- and 32-bit systems (by emulating a 64-bit datum with a class). `SPARCMMU` is only dependent upon SIND running in the same address space as the target process. Much of the rest of the code is supporting classes for the interpreter (`SPARCInstruction`, etc) or data structure classes (`AddressHash`, etc), and is not specific to any platform. To avoid the annoying unknown bit-width of `int` problem, all the code that needs to specify a given bit width uses the POSIX typing standard (i.e. `uint32_t` for a 32-bit unsigned int).

Appendix A

Technical Details

This appendix describes the nitty-gritty details necessary to take the current SIND code base and build it, or (hopefully) extend or debug it.

A.1 Source Layout and Directory Organization

The base directory contains all the standard GNU files, as well as an out of date configuration script. The `doc` directory contains documents related to SIND, in particular this thesis and the technical reports written about it. All the code is located in the `src` directory. All of the base classes are in `src` the class definitions are in `include` and the implementation files are in `src`. `src` also includes a simple makefile for building these classes. Off of this directory are directories for each architecture. Currently there are only two `PowerPC` and `SPARC`. `PowerPC` contains Dino Dai Zovi's beginnings of a PowerPC interpreter, and `SPARC` contains all of my SPARC and Solaris specific code. The architecture directories are organized similar to the base class directory, class definitions and other header files are in `include`, and the implementations as well as the Makefile are in the `SPARC` directory.

A.2 Where Functionality Resides

In order to get a good handle on the code base it is necessary to create a mapping between the modules described in the design section and the actual files in the directories.

Include Files	
AddressHash.h	definition of the AddrHash data structure
bootstrap.h	function prototypes for bootstrapping
driver.h	definition for driver program
signal_handling.h	prototypes of internal signal handling functions
signal_stuff.h	prototypes of superposed signal functions
SPARCCPU.h	definition of the SPARCCPU (interpreter) class
SPARCDispatch.h	definition of the SPARCDispatch class
SPARCExceptions.h	definitions of error conditions in the interpreter
SPARCFPU.h	definition of the floating point unit
SPARCFragCache.h	definition of the SPARCFragCache class
SPARCInstruction.h	the SPARC instruction class
SPARCInstrFmt.h	a wrapper for a 32-bit integer to readily parse it as an instruction
SPARCMMU.h	definition of the SPARC MMU class
SPARCRegisters.h	definition of registers (both general purpose and special)
SPARCTrace.h	definition of the data structure used to hold traces
SPARCTransformer.h	interface class for transformers
SPARCTrap.h	definition of trap handling functions (for syscall manager)

Table A.1: Include files to modules

Files and Modules

Module	File(s)
Bootstrap	bootstrap.h, bootstrap.c
Interpreter	SPARCCPU.*, SPARCExceptions.h, SPARCFPU.*,SPARCInstrFmt.h, SPARCInstruction.h, SPARCRegisters.h
Trap/Syscall Manger	SPARCTrap.*
Dispatch	SPARCDispatch.*
Transformer	anything implementing SPARCTransformer
Memory Manager	SPARCMMU.*
Fragment Cache	SPARCFragCache.*

Table A.2: modules' source files

A.3 Compilation and Architecture Support

A.3.1 Makefiles

Compilation is currently managed by a set of Makefiles. Each directory that has compilable source will have its own Makefile. The main one for the experimental system is the Makefile in the SPARC directory. This Makefile follows the convention of having the `all` and `clean` targets. `all` will build everything (including files in parent directories) and link it together to create the SIND shared object. After building there should be a file named `libsind.so` in the directory. This is SIND.

A.3.2 Compilation Flags

There are several compilation flags that guide the preprocessing of the SIND source. The first is `SIND_ARCH64`. If this is defined, then it is assumed that the SIND code is running on a 64-bit machine and so can use native 64-bit integers. If this is not defined, then

emulation of 64-bit wide data is done through a class (DoubleWord) using overloaded operators. The second flag is `DEBUG`; if this is defined, scores of debugging statements will be compiled into SIND. This generates voluminous output, and should only be used for debugging. If not defined, SIND's printouts will be limited to a few lines of text when starting up. Also, this means that any extensions made to SIND should respect this convention and enclose debugging statements in `#ifdef DEBUG` conditionals. The next flag of note is `TRACING`, if defined the interpreter will be compiled with tracing support, otherwise all code will be executed in the interpreter. This flag was introduced to allow me to debug interpreter errors even after I had implemented a tracing infrastructure. Another notable flag is `TIMING`; if defined, then SIND will time itself and print out the results. Currently, timing code exists only in the interpreter, and this times the execution in the main interpreter loop (`executeLoop()`). Lastly, there's the `BOGUS` flag, this flag should only be set if compiling a BOGUS version of SIND. The bogus version is used to get a highly accurate snapshot of the machine state for debugging purposes and is discussed in more detail in section 6.2.

A.3.3 Supported Architectures

All the platform independent code should work on any 64-bit platform, and includes enough infrastructure that it could readily be made to work on any 32-bit platform. All the platform dependent code will only work on an UltraSPARC running Solaris2.x and up.

References

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[2] Derek Bruening and Saman Amarasinghe. The DynamoRIO Collaboration. http://www.cag.lcs.mit.edu/dynamorio/.

[3] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Yadavalli, and J. Yates. FX!32 a profile-directed binary translator, 1998.

[4] Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zang. Automatic detection and prevention of buffer-overflow attacks. *7th USENIX Security Symposium*, 1998.

[5] Kemal Ebcioğlu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA*, pages 26–37, 1997.

[6] Richard McDougall Jim Mauro. *Solaris Internals: Core Kernel Architecture*. Prentice Hall PTR, first edition, 5 October 2000.

[7] A. Klaiber. The technology behind Crusoe processors, 2000.

[8] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, first edition, 15 January 2000.

[9] Sun Microsystems. proc - /proc, the process file system. In *SunOS 5.8 Manual*, chapter 4.

[10] Sun Microsystems. ptrace - allows a parent process to control the execution of a child process. In *SunOS 5.8 Manual*, chapter 2.

[11] Sun Microsystems. Solaris linker and libraries guide. Online PDF manual, Part Number 816-0559-10.

[12] Julian Seward and Nick Nethercote. Valgrind, an open-source memory debugger for x86-linux. http://developer.kde.org/ sewardj.

Appendix C

Randomized Instruction Set Emulation

ELENA GABRIELA BARRANTES, DAVID H. ACKLEY, STEPHANIE FORREST, and DARKO STEFANOVIĆ
University of New Mexico

Injecting binary code into a running program is a common form of attack. Most defenses employ a "guard the doors" approach, blocking known mechanisms of code injection. *Randomized instruction set emulation* (RISE) is a complementary method of defense, one that performs a hidden randomization of an application's machine code. If foreign binary code is injected into a program running under RISE, it will not be executable because it will not know the proper randomization. The paper describes and analyzes RISE, describing a proof-of-concept implementation built on the open-source Valgrind IA32-to-IA32 translator. The prototype effectively disrupts binary code injection attacks, without requiring recompilation, linking, or access to application source code. Under RISE, injected code (attacks) essentially executes random code sequences. Empirical studies and a theoretical model are reported which treat the effects of executing random code on two different architectures (IA32 and PowerPC). The paper discusses possible extensions and applications of the RISE technique in other contexts.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*; D.3.4 [**Programming Languages**]: Processors—*Interpreters, runtime environments*

General Terms: Security

Additional Key Words and Phrases: Automated diversity, randomized instruction sets, software diversity

An earlier version of this paper was published as Barrantes, E. G., Ackley, D. H., Forrest, S., Palmer, T. S., Stefanović, D., and Dai Zovi, D. 2003. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pp. 281–289. This version adds a detailed model and analysis of the safety of random bit execution, and presents additional empirical results on the prototype's effectiveness and performance.
The authors gratefully acknowledge the partial support of the National Science Foundation (grants ANIR-9986555, CCR-0219587, CCR-0085792, CCR-0311686, EIA-0218262, EIA-0238027, and EIA-0324845), the Office of Naval Research (grant N00014-99-1-0417), Defense Advanced Research Projects Agency (grants AGR F30602-00-2-0584 and F30602-02-1-0146), Sandia National Laboratories, Hewlett-Packard gift 88425.1, Microsoft Research, and Intel Corporation. Any opinions, findings, conclusions, or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.
Authors' address: Department of Computer Science, University of New Mexico, MSC01 1130, Albuquerque, NM 87131-1386.
Stephanie Forrest is also with the Santa Fe Institute, 1399 Hyde Park Rd, Santa Fe, NM 87501.

1. INTRODUCTION

Standardized machine instruction sets provide consistent interfaces between software and hardware, but they are a double-edged sword. Although they yield great productivity gains by enabling independent development of hardware and software, the ubiquity of well-known instructions sets also allows a single attack designed around an exploitable software flaw to gain control of thousands or millions of systems. Such attacks could be stopped or greatly hindered if each protected system could be economically destandardized, so that a different attack would have to be created specifically for each new target, using information that was difficult or impossible for an outsider to obtain. The automatic diversification we explore in this paper is one such destandardization technique.

Many existing defenses against machine code injection attacks block the known routes by which foreign code is placed into a program's execution path. For example, stack defense mechanisms [Chiueh and Hsu 2001; Cowan et al. 1998; Etoh and Yoda 2000, 2001; Forrest et al. 1997; Frantzen and Shuey 2001; Nebenzahl and Wool 2004; Prasad and Chiueh 2003; Vendicator 2000; Xu et al. 2002] protect return addresses and defeat large classes of buffer overflow attacks. Other mechanisms defend against buffer overflows elsewhere in program address space [PaX Team 2003], against alternative overwriting methods [Cowan et al. 2001], or guard from known vulnerabilities through shared interfaces [Avijit et al. 2004; Baratloo et al. 2000; Lhee and Chapin 2002; Tsai and Singh 2001]. Our approach is functionally similar to the PAGEEXEC feature of PaX [PaX Team 2003], an issue we discuss in Section 6.

Rather than focusing on any particular code injection pathway, a complementary approach would disrupt the operation of the injected code itself. In this paper we describe *randomized instruction set emulation* (RISE), which uses a machine emulator to produce automatically diversified instruction sets. With such instruction set diversification, each protected program has a different and secret instruction set, so that even if a foreign attack code manages to enter the execution stream, with very high probability the injected code will fail to execute properly.

In general, if there are many possible instruction sets compared to the number of protected systems and the chosen instruction set in each case is externally unobservable, different attacks must be crafted for each protected system and the cost of developing attacks is greatly increased. In RISE, each byte of protected program code is scrambled using pseudorandom numbers seeded with a random key that is unique to each program execution. Using the scrambling constants it is trivial to recover normal instructions executable on the physical machine, but without the key it is infeasible to produce even a short code sequence that implements any given behavior. Foreign binary code that manages to reach the emulated execution path will be descrambled without ever having been correctly scrambled, foiling the attack, and producing pseudorandom code that will usually crash the protected program.

1.1 Threat Model

The set of attacks that RISE can handle is slightly different from that of many defense mechanisms, so it is important to identify the RISE threat model clearly.

Our specific threat model is *binary code injection from the network into an executing program*. This includes many real-world attack mechanisms, but explicitly excludes several others, including the category of attacks loosely grouped under the name "return into libc" [Nergal 2001] which modify data and addresses so that code already existing in the program is subverted to execute the attack. These attacks might or might not use code injection as part of the attack. Most defenses against code injection perform poorly against this category as it operates at a different level of abstraction; complementary defense techniques are needed, and have been proposed, such as address obfuscation [Bhatkar et al. 2003; Chew and Song 2002; PaX Team 2003], which hide and/or randomize existing code locations or interface access points.

The restriction to code injection attacks excludes data only attacks such as nonhybrid versions of the "return into libc" class mentioned above, while focusing on binary code excludes attacks such as macro viruses that inject code written in a higher-level language. Finally, we consider only attacks that arrive via network communications and therefore we treat the contents of local disks as trustworthy before an attack has occurred.

In exchange for these limitations, RISE protects against all binary code injection attacks, regardless of the method by which the machine code is injected. By defending the code itself, rather than any particular access route into the code, RISE offers the potential of blocking attacks based on injection mechanisms that have yet to be discovered or revealed.

This threat model is related to, but distinct from, other models used to characterize buffer overflow attacks [Cowan et al. 2000, 2001]. It includes any attack in which native code is injected into a running binary, even by means that are not obviously buffer overflows, such as misallocated malloc headers, footer tags [Security Focus 2003; Xu et al. 2003], and format string attacks that write a byte to arbitrary memory locations [Gera and Riq 2002; Newsham 2000]. RISE protects against injected code arriving by any of these methods. On the other hand, other defense mechanisms, such as the address obfuscation mentioned above, can prevent attacks that are specifically excluded from our code injection threat model.

We envision the relatively general code-based mechanism of RISE being used in conjunction with data and address diversification-based mechanisms to provide deeper, more principled, and more robust defenses against both known and unknown attacks.

1.2 Overview

This paper describes a proof-of-concept RISE system, which builds randomized instruction set support into a version of the Valgrind IA32-to-IA32 binary translator [Nethercote and Seward 2003; Seward and Nethercote 2004]. Section 2 describes a randomizing loader for Valgrind that scrambles code sequences loaded into emulator memory from the local disk using a hidden random key. Then, during Valgrind's emulated instruction fetch cycle, fetched instructions are unscrambled, yielding the unaltered IA32 machine code sequences of the protected application. The RISE design makes few demands on the supporting

emulator and could be easily ported to any binary-to-binary translator for which source code is available.

Section 3 reports empirical tests of the prototype and confirms that RISE successfully disrupts a range of actual code injection attacks against otherwise vulnerable applications. In addition, it highlights the extreme fragility of typical attacks and comments on performance issues.

A basic property of the RISE defense mechanism is that if an attack manages to inject code by any means, essentially random machine instructions will be executed. Section 4 investigates the likely effects of such an execution in several different execution contexts. Experimental results are reported and theoretical analyses are given for two different architectures. There is always a possibility that random bits could create valid instructions and instruction sequences. We present empirical data suggesting that the majority of random code sequences will produce an address fault or illegal instruction quickly, causing the program to abort. Most of the remaining cases throw the program into a loop, effectively stopping the attack. Either way, an attempted takeover is downgraded into a denial-of-service attack against the exploitable program.

Unlike compiled binary code, which uses only a well-defined and often relatively small selection of instructions, random code is unconstrained. The behavior of random code execution in the IA32 architecture can involve the effects of undocumented instructions and whatever instruction set extensions (e.g., MMX, SSE, and SSE2) are present, as well as the effects of random branch offsets combined with multibyte, variable-length instructions. Although those characteristics complicate a tight theoretical analysis of random bit executions on the IA32, models for more constrained instruction set architectures, such as the PowerPC, lead to a closer fit to the observed data.

Section 6 summarizes related work, Section 7 discusses some of the implications and potential vulnerabilities of the RISE approach, and Section 8 concludes the paper.

2. TECHNICAL APPROACH AND IMPLEMENTATION

This section describes the prototype implementation of RISE using Valgrind [Nethercote and Seward 2003; Seward and Nethercote 2004] for the Intel IA32 architecture. Our strategy is to provide each program copy its own unique and private instruction set. To do this, we consider what is the most appropriate machine abstraction level, how to scramble and descramble instructions, when to apply the randomization and when to descramble, and how to protect interpreter data. We also describe idiosyncrasies of Valgrind that affected the implementation.

2.1 Machine Abstraction Level

The native instruction set of a machine is a promising computational level for automated diversification because all computer functionality can be expressed in machine code. This makes the machine-code level desirable to attack and protect. However, automated diversification is feasible at higher levels of abstraction, although there are important constraints on suitable candidates.

Language diversification seems most promising for languages that are interpreted or executed directly by a virtual machine. Randomizing source code for a compiled language would protect only against injections at compile time. An additional constraint is the possibility of crafting attacks at the selected language level. Although it is difficult to evaluate this criterion in the abstract, we could simply choose languages for which those attacks have already been shown to exist, such as Java, Perl, and SQL [Harper 2002]. And in fact, proposals for diversifying these higher levels have been made [Boyd and Keromytis 2004; Kc et al. 2003]. Macro languages provide another example of a level that could be diversified to defeat macro viruses.

Finally, it is necessary to have a clear trust boundary between internal and external programs so that it is easy to decide which programs should be randomized. The majority of programs should be internal to the trust boundary, or the overhead of deciding what is trusted and untrusted will become too high. This requirement eliminates most web-client scripting languages such as Javascript because a user decision about trust would be needed every time a Javascript program was going to be executed on a client. A native instruction set, with a network-based threat model, provides a clear trust boundary, as all legitimately executing machine code is stored on a local disk.

An obvious drawback of native instruction sets is that they are traditionally physically encoded and not readily modifiable. RISE therefore operates at an intermediate level, using software that performs binary-to-binary code translation. The performance impact of such tools can be minimal [Bala et al. 2000; Bruening et al. 2001]. Indeed, binary-to-binary translators sometimes improve performance compared to running the programs directly on the native hardware [Bala et al. 2000].

For ease of research and dissemination, we selected the open-source emulator, Valgrind, for our prototype. Although Valgrind is described primarily as a tool for detecting memory leaks and other program errors, it contains a complete IA32-to-IA32 binary translator. The primary drawback of Valgrind is that it is very slow, largely owing to its approach of translating the IA32 code into an intermediate representation and its extensive error checking. However, the additional slowdown imposed by adding RISE to Valgrind is modest, and we are optimistic that porting RISE to a more performance-oriented emulator would yield a fully practical code defense.

2.2 Instruction Set Randomization

Instruction set randomization could be as radical as developing a new set of opcodes, instruction layouts, and a key-based toolchain capable of generating the randomized binary code. And, it could take place at many points in the compilation-to-execution spectrum. Although performing randomization early could help distinguish code from data, it would require a full compilation environment on every machine, and recompiled randomized programs would likely have one fixed key indefinitely. RISE randomizes as late as possible in the process, scrambling each byte of the trusted code as it is loaded into the emulator, and then unscrambling it before execution. Deferring the randomization to

load time makes it possible to scramble and load existing files in the executable and linking format (ELF) [Tool Interface Standards Committee . 1995] directly, without recompilation or source code, provided we can reliably distinguish code from data in the ELF file format.

The unscrambling process needs to be fast, and the scrambling process must be as hard as possible for an outsider to deduce. Our current default approach is to generate at load time a pseudorandom sequence the length of the overall program text using the Linux `/dev/urandom` device [Tso 1998], which uses a secret pool of true randomness to seed a pseudorandom stream generated by feedback through SHA1 hashing. The resulting bytes are simply XORed with the instruction bytes to scramble and unscramble them. In addition, it is possible to specify the length of the key, and a smaller key can be tiled over the process code. If the underlying truly random key is long enough, and as long as it is infeasible to invert SHA1 [Schneier 1996], we can be confident that an attacker cannot break the entire sequence. The security of this encoding is discussed further in Section 7.

2.3 Design Decisions

Two important aspects of the RISE implementation are how it handles shared libraries and how it protects the plaintext executable.

Much of the code executed by modern programs resides in shared libraries. This form of code sharing can significantly reduce the effect of the diversification, as processes must use the same instruction set as the libraries they require. When our load-time randomization mechanism writes to memory that belongs to shared objects, the operating system does a copy-on-write, and a private copy of the scrambled code is stored in the virtual memory of the process. This significantly increases memory requirements, but increases interprocess diversity and avoids having the plaintext code mapped in the protected processes' memory. This is strictly a design decision, however. If the designer is willing to sacrifice some security, it can be arranged that processes using RISE share library keys, and so library duplication could be avoided.

Protecting the plaintext instructions inside Valgrind is a second concern. As Valgrind simulates the operation of the CPU, during the fetch cycle when the next byte(s) are read from program memory, RISE intercepts the bytes and unscrambles them; the scrambled code in memory is never modified. Eventually, however, a plaintext piece of the program (semantically equivalent to the block of code just read) is written to Valgrind's cache. From a security point of view, it would be best to separate the RISE address space completely from the protected program address space, so that the plaintext is inaccessible from the program, but as a practical matter this would slow down emulator data accesses to an extreme and unacceptable degree. For efficiency, the interpreter is best located in the same address space as the target binary, but of course this introduces some security concerns. A RISE-aware attacker could aim to inject code into a RISE data area, rather than that of the vulnerable program. This is a problem because the cache cannot be encrypted. To protect the cache its pages are kept as read-and-execute only. When a new translated basic block is ready to be

written to the cache, we mark the affected pages as writable, execute the write action, and restore the pages to their original nonwritable permissions. A more principled solution would be to randomize the location of the cache and the fragments inside it, a possibility for future implementations of RISE.

2.4 Implementation Issues

Our current implementation does not handle self-modifying code, but it has a primitive implementation of an interface to support dynamically generated code. We consider arbitrary self-modifying code as an undesirable programming practice and agree with Valgrind's model of not allowing it. However, it is desirable to support legitimate dynamically generated code, and we intend to provide eventually a complete interface for this purpose.

An emulator needs to create a clear boundary between itself and the process to be emulated. In particular, the emulator should not use the same shared libraries as the process being emulated. Valgrind deals with this issue by adding its own implementation of all library functions it uses, with a local modified name for example, `VGplain_printf` instead of `printf`. However, we discovered that Valgrind occasionally jumped into the target binary to execute low-level functions (e.g., `__umoddi` and `__udivdi`). When that happened, the processor attempted to execute instructions that had been scrambled for the emulated process, causing Valgrind to abort. Although this was irritating, it did demonstrate the robustness of the RISE approach in that these latent boundary crossings were immediately detected. We worked around these dangling unresolved references by adding more local functions to Valgrind and renaming affected symbols with local names (e.g., `rise_umoddi` instead of "%" (the modulo operator)).

A more subtle problem arises because the IA32 does not impose any data and code separation requirement, and some compilers insert dispatch tables directly in the code. In those cases, the addresses in such internal tables are scrambled at load time (because they are in a code section), but are not descrambled at execution time because they are read as data. Although this does not cause an illegal operation, it causes the emulated code to jump to a random address and fail inappropriately. At interpretation time, RISE looks for code sequences that are typical for jump-table referencing and adds machine code to check for in-code references into the block written to the cache. If an in-code reference is detected when the block is executing, our instrumentation descrambles the data that was retrieved and passes it in the clear to the next (real) instruction in the block. This scheme could be extended to deal with the general case of using code as data by instrumenting every dereference to check for in-code references. However, this would be computationally expensive, so we have not implemented it in the current prototype. Code is rarely used as data in legitimate programs except in the case of virtual machines, which we address separately.

An additional difficulty was discovered with Valgrind itself. The thread support implementation and the memory inspection capabilities require Valgrind to emulate itself at certain moments. To avoid infinite emulation regress, it has a special workaround in its code to execute some of its own functions natively during this self-emulation. We handled this by detecting Valgrind's own address

ranges and treating them as special cases. This issue is specific to Valgrind, and we expect not to encounter it in other emulators.

3. EFFICACY AND PERFORMANCE OF RISE

The results reported in this section were obtained using the RISE prototype, available under the GPL from `http://cs.unm.edu/~immsec`. We have tested RISE's ability to run programs successfully under normal conditions and its ability to disrupt a variety of machine code injection attacks. The attack set contained 20 synthetic and 15 real attacks.

The synthetic attacks were obtained from two sources. Two attacks, published by Fayolle and Glaume [2002], create a vulnerable buffer—in one case on the heap and in the other case on the stack—and inject shellcode into it. The remaining 18 attacks were executed with the attack toolkit provided by Wilander and Kamkar and correspond to their classification of possible buffer overflow attacks [Wilander and Kamkar 2003] according to technique (direct or pointer redirection), type of location (stack, heap, BSS, or data segment), and attack target (return address, old base pointer, function pointer, and longjump buffer). Without RISE, either directly on the processor or using Valgrind, all of these attacks successfully spawn a shell. Using RISE, the attacks are stopped.

The real attacks were launched from the CORE impact attack toolkit [CORE Security 2004]. We selected 15 attacks that satisfied the following requirements of our threat model and the chosen emulator: the attack is launched from a remote site; the attack injects binary code at some point in its execution; and, the attack succeeds on a Linux OS. Because Valgrind runs under Linux; we focused on Linux distributions, reporting data from Mandrake 7.2 and versions of RedHat from 6.2 to 9.

3.1 Results

All real (nonsynthetic) attacks were tested on the vulnerable applications before retesting with RISE. All of them were successful against the vulnerable services without RISE, and they were all defeated by RISE (Table I).

Based on the advisories issued by CERT in the period between 1999 and 2003, Xu et al. [2003] classify vulnerabilities that can inject binary code into a running process according to the method used to modify the execution flow: buffer overflows, format string vulnerabilities, malloc/free, and integer manipulation errors. Additionally, the injected code can be placed in different sections of the process (stack, heap, data, BSS). The main value of RISE is its imperviousness to the entry method and/or location of the attack code, as long as the attack itself is expressed as binary code. This is illustrated by the diversity of vulnerability types and shellcode locations used in the real attacks (columns 3 and 4 of Table I).

The available synthetic attacks are less diverse in terms of vulnerability type. They are all buffer overflows. However, they do have attack code location variety (stack, heap, and data), and more importantly, they have controlled diversity of corrupted code address types (return address, old base pointer, function pointer, and longjump buffer as either local variable or parameter),

Table I. Results of Attacks Against Real Applications Executed under RISE

Attack	Linux Distribution	Vulnerability	Location of Injected Code	Stopped by RISE
Apache OpenSSL SSLv2	RedHat 7.0 & 7.2	Buffer overflow and malloc/free	Heap	√
Apache mod php	RedHat 7.2	Buffer overflow	Heap	√
Bind NXT	RedHat 6.2	Buffer overflow	Stack	√
Bind TSIG	RedHat 6.2	Buffer overflow	Stack	√
CVS flag insertion heap exploit	RedHat 7.2 & 7.3	Malloc/free	Heap	√
CVS pserver double free	RedHat 7.3	Malloc/Free	Heap	√
PoPToP Negative Read	RedHat 9	Integer error	Heap	√
ProFTPD _xlate_ascii _write off-by-two	RedHat 9	Buffer overflow	Heap	√
rpc.statd format string	RedHat 6.2	Format string	GOT	√
SAMBA nttrans	RedHat 7.2	Buffer overflow	Heap	√
SAMBA trans2	RedHat 7.2	Buffer overflow	Stack	√
SSH integer overflow	Mandrake 7.2	Integer error	Stack	√
sendmail crackaddr	RedHat 7.3	Buffer overflow	Heap	√
wuftpd format string	RedHat 6.2–7.3	Format string	Stack	√
wuftpd glob "~{"	RedHat 6.2–7.3	Buffer overflow	Heap	√

Column 1 gives the exploit name (and implicitly the service against which it was targeted).
The vulnerability type and attack code (shellcode) locations are included (columns 3 and 4, respectively).
The result of the attack is given in column 5.

and offer either direct or indirect execution flow hijacking (see Wilander and Kamkar [2003]). All of Wilander's attacks have the shellcode located in the data section. Both of Fayolle and Glaume's exploits use direct return address pointer corruption. The stack overflow injects the shellcode on the stack, and the heap overflow locates the attack code on the heap. All synthetic attacks are successful (spawn a shell) when running natively on the processor or over unmodified Valgrind. All of them are stopped by RISE (column 5 of Table II).

When we originally tested real attacks and analyzed the logs generated by RISE, we were surprised to find that nine of them failed without ever executing the injected attack code. Further examination revealed that this was due to various issues with Valgrind itself, which have been remedied in later versions. The current RISE implementation in Valgrind 2.0.0 does not have this behavior. All attacks (real and synthetic) are able to succeed when the attacked program runs over Valgrind, just as they do when running natively on the processor.

These results confirm that we successfully implemented RISE and that a randomized instruction set prevents injected machine code from executing, without the need for any knowledge about how or where the code was inserted in process space.

3.2 Performance

Being emulation based, RISE introduces execution costs that affect application performance. For a proof-of-concept prototype, correctness and defensive power were our primary concerns, rather than minimizing resource overhead. In this section, we describe the principal performance costs of the RISE approach,

Table II. Results of the Execution of Synthetic Attacks under RISE

Type of Overflow	Shellcode Location	Exploit Origin	Number of ≠ Pointer Types	Stopped by RISE
Stack direct	Data	Wilander and Kamkar [2003]	6	6 (100%)
Data direct	Data	Wilander and Kamkar [2003]	2	2 (100%)
Stack indirect	Data	Wilander and Kamkar [2003]	6	6 (100%)
Data indirect	Data	Wilander and Kamkar [2003]	4	4 (100%)
Stack direct	Stack	Fayolle and Glaume [2002]	1	1 (100%)
Stack direct	Heap	Fayolle and Glaume [2002]	1	1 (100%)

Type of overflow (column 1) denotes the location of the overflowed buffer (stack, heap or data) and the type of corruption executed: *direct* modifies a code pointer during the overflow (such as the return address), and *indirect* modifies a data pointer that eventually is used to modify a code pointer.
Shellcode location (column 2) indicates the segment where the actual malicious code was stored.
Exploit origin (column 3) gives the paper from which the attacks were taken.
The number of pointer types (column 4) defines the number of different attacks that were tried by varying the type of pointer that was overflowed.
Column 5 gives the number of different attacks in each class that were stopped by RISE.

which include a once-only time cost for code randomization during loading, time for derandomization while the process executes, and space overheads.

Although in the following we assume an all-software implementation, RISE could also be implemented with hardware support, in which case we would expect much better performance because the coding and decoding could be performed directly in registers rather than executing two different memory accesses for each fetch.

The size of each RISE-protected process is increased because it must have its own copy of any library it uses. Moreover, the larger size is as much as doubled to provide space for the randomization mask.[1]

A software RISE uses dynamic binary translation, and pays a runtime penalty for this translation. Valgrind amortizes interpretation cost by storing translations in a cache, which allows native-speed execution of previously interpreted blocks.

Valgrind is much slower than binary translators [Bala et al. 2000; Bruening et al. 2001] because it converts the IA32 instruction stream into an intermediate representation before creating the code fragment. However, we will give some evidence that long-running, server-class processes can execute at reasonable speeds and these are precisely the ones for which RISE is most needed.

As an example of this effect, Table III provides one data point about the long-term runtime costs of using RISE, using the Apache web server in the face of a variety of nonattack workloads. Classes 0 to 3, as defined by SPEC Inc. [1999], refer to the size of the files that are used in the workload mix. Class 0 is the least I/O intensive (files are less than 1 KB long), and class 3 is the one that uses the most I/O (files up to 1000 KB long). As expected, on I/O bound mixes, the throughput of Apache running over RISE is closer to Apache running

[1]A RISE command-line switch controls the length of the mask, which is then tiled to cover the program. A 1000-byte mask, for example, would be a negligible cost for mask space, and very probably would provide adequate defense. In principle, however, it might open a within-run vulnerability owing to key reuse.

Table III. Comparison of the Average Time Per Operation between Native Execution of Apache and Apache over RISE

	Native Execution		Execution over RISE		RISE/
Mix Type	Mean (ms)	Std. Dev.	Mean (ms)	Std. Dev.	Native
Class 0	177.32	422.22	511.73	1,067.79	2.88
Class 1	308.76	482.31	597.11	1,047.23	1.93
Class 2	1,230.75	624.58	1,535.24	1,173.57	1.25
Class 3	10,517.26	3,966.24	11,015.74	4,380.26	1.05
Total	493.80	1,233.56	802.63	1,581.50	1.62

Presented times were obtained from the second iteration in a standard SPECweb99 configuration (300 s warm up and 1200 s execution).

directly on the processor.[2] Table III shows that the RISE prototype slows down by a factor of no more than 3, and sometimes by as little as 5%, compared with native execution, as observed by the client. These results should not be taken as a characterization of RISE's performance, but as evidence that cache-driven amortization and large I/O and network overheads make the CPU performance hit of emulation just one (and possibly not the main) factor in evaluating the performance of this scheme.

By contrast, short interactive jobs are more challenging for RISE performance, as there is little time to amortize mask generation and cache filling. For example, we measured a slowdown factor of about 16 end-to-end when RISE protecting all the processes invoked to make this paper from LaTeX source.

Results of the Dynamo project suggest that a custom-built dynamic binary translator can have much lower overheads than Valgrind, suggesting that a commercial-grade RISE would be fast enough for widespread use; in long-running contexts where performance is less critical, even our proof-of-concept prototype might be practical.

4. RISE SAFETY: EXPERIMENTS

Code diversification techniques such as RISE rely on the assumption that random bytes of code are highly unlikely to execute successfully. When binary code is injected by an attacker and executes, it is first derandomized by RISE. Because the attack code was never prerandomized, the effect of derandomizing is to transform the attack code into a random byte string. This is invisible to the interpretation engine, which will attempt to translate, and possibly execute, the string. If the code executes at all, it clearly will not have the effect intended by the attacker. However, there is some chance that the random bytes might correspond to an executable sequence, and an even smaller chance that the executed sequence of random bytes could cause damage. In this section, we measure the likelihood of these events under several different assumptions, and in the following section we develop theoretical estimates.

Our approach is to identify the possible actions that randomly formed instructions in a sequence could perform and then to calculate the probabilities

[2]The large standard deviations are typical of SPECweb99, as web server benchmarks have to model long-tailed distributions of request sizes [Nahum 2002; SPEC Inc. 1999].

for these different events. There are several broad classes of events that we consider: illegal instructions that lead to an error signal, valid execution sequences that lead to an infinite loop or a branch into valid code, and other kinds of errors. There are several subtle complications involved in the calculations, and in some cases we make simplifying assumptions. The simplifications lead to a conservative estimate of the risk of executing random byte sequences.

4.1 Possible Behaviors of Random Byte Sequences

First, we characterize the possible events associated with a generic processor or emulator attempting to execute a random symbol. We use the term *symbol* to refer to a potential execution unit, because a symbol's length in bytes varies across different architectures. For example, instruction length in the PowerPC architecture is exactly 4 bytes and in the IA32 it can vary between 1 and 17 bytes. Thus, we adopt the following definitions:

(1) A *symbol* is a string of l bytes, which may or may not belong to the instruction set. In a RISC architecture, the string will always be of the same length, while for CISC it will be of variable length.
(2) An *instruction* is a symbol that belongs to the instruction set.

In RISE there is no explicit recognition of an attack, and success is measured by how quickly and safely the attacked process is terminated. Process termination occurs when an error condition is generated by the execution of random symbols. Thus, we are interested in the following questions:

(1) How soon will the process *crash* after it begins executing random symbols? (Ideally, in the first symbol.)
(2) What is the probability that an execution of random bytes will branch to valid code or enter an infinite loop (*escape*)? (Ideally, 0.)

Figure 1 illustrates the possible outcomes of executing a single random symbol. There are three classes of outcome: an error that generates a signal, a branch into executable memory in the process space that does not terminate in an error signal (which we call *escape*), and the simple execution of the symbol with the program pointer moving to the next symbol in the sequence. Graph traversal always begins in the *start* state, and proceeds until a terminating node is reached (*memory error signal*, *instruction-specific error signal*, *escape*, or *start*).

The term *crash* refers to any error signal (the states labeled *invalid opcode*, *specific error signal*, and *memory error signal* in Figure 1). Error signals do not necessarily cause process termination due to error, because the process could have defined handlers for some of the error signals. We assume, however, that protected processes have reasonable signal handlers, which terminate the process after receiving such a signal. We include this outcome in the event *crash*.

The term *escape* describes a branch from the sequential flow of execution inside the random code sequence to any executable memory location. This event occurs when the instruction pointer (IP) is modified by random instructions to

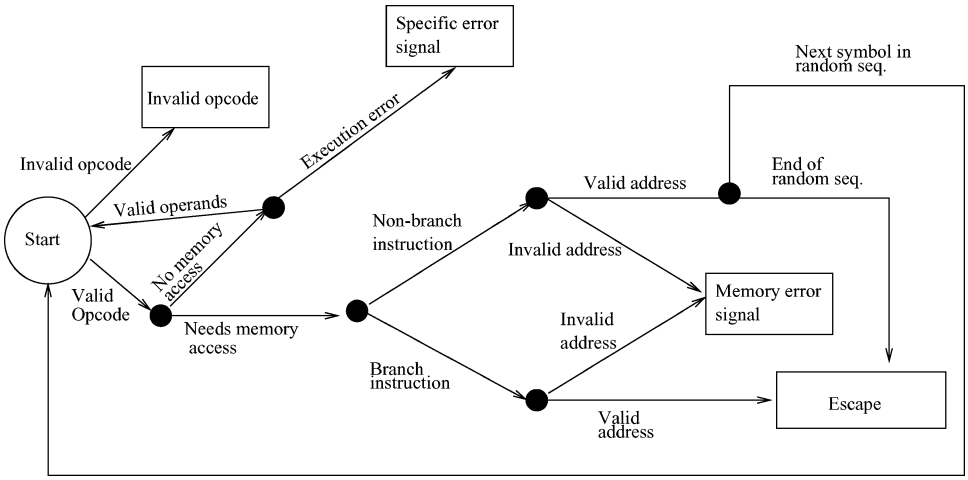

Fig. 1. State diagram for random code execution. The graph depicts the possible outcomes of executing a single random symbol. For variable-length instruction sets; the start state represents the reading of bytes until a nonambiguous decision about the identity of the symbol can be made.

point either to a location inside the executable code of the process, or to a location in a data section marked as executable even if it does not typically contain code.

An *error signal* is generated when the processor attempts to decode or execute a random symbol in the following cases:

(1) *Illegal instruction*: The symbol has no further ambiguity and it does not correspond to a defined instruction. The persymbol probability of this event depends solely on the density of the instruction set. An illegal instruction is signaled for undefined opcodes, illegal combinations of opcode and operand specifications, reserved opcodes, and opcodes undefined for a particular configuration (e.g., a 64-bit instruction on a 32-bit implementation of the PowerPC architecture).

(2) *Illegal read/write*: The instruction is legal, but it attempts to access a memory page to which it does not have the required operation privileges, or the page is outside the process' virtual memory.

(3) *Operation error*: Execution fails because the process state has not been properly prepared for the instruction; for example, division by 0, memory errors during a string operation, accessing an invalid port, or invoking a nonexistent interrupt.

(4) *Illegal branch*: The instruction is of the control transfer type and attempts to branch into a nonexecutable or nonallocated area.

(5) *Operation not permitted*: A legal instruction fails because the rights of the owner process do not allow its execution, for example, an attempt to use a privileged instruction in user mode.

There are several complications associated with branch instructions, depending on the target address of the branch. We assume that the only dangerous class of branch is a correctly invoked system call. The probability of randomly

invoking a system call in Linux is $\frac{1}{256} \times \frac{1}{256} \approx 1.52 \times 10^{-5}$ for IA32, and at most $\frac{1}{2^{32}} \approx 2.33 \times 10^{-10}$ for the 32-bit PowerPC. This is without adding the restriction that the arguments be reasonable. Alternatively, a process failure could remain hidden from an external observer, and we will see that this event is more likely.

A branch into the executable code of the process (ignoring alignment issues) will likely result in the execution of at least some instructions, and will perhaps lead to an infinite loop. This is an undesirable event because it hides the attack attempt even if it does not damage permanent data structures. We model successful branches into executable areas (random or nonrandom) as always leading to the *escape* state in Figure 1. This conservative assumption allows us to estimate how many attack instances will not be immediately detected. These "escapes" do not execute hostile code. They are simply attack instances that are likely not to be immediately observed by an external process monitor. The probability of a branch resulting in a crash or an escape depends at least in part on the size of the executing process, and this quantity is a parameter in our calculations.

Different types of branches have different probabilities of reaching valid code. For example, if a branch has the destination specified as a full address constant (*immediate*) in the instruction itself, it will be randomized, and the probability of landing in valid code will depend only on the density of valid code in the total address space, which tends to be low. A return takes the branching address from the current stack pointer, which has a high probability of pointing to a real-process return address.

We model these many possibilities by dividing memory accesses, for both branch and nonbranch instructions into two broad classes:

(1) *Process-state dominated*: When the randomized exploit begins executing, the only part of the process that has been altered is the memory that holds the attack code. Most of the process state (e.g., the contents of the registers, data memory, and stack) remains intact and consistent. However, we do not have good estimates of the probability that using these values from registers and memory will cause an error. So, we arbitrarily assign probabilities for these values and explore the sensitivity of the system to different probabilities. Experimentally we know that most memory accesses fail (see Figure 2).

(2) *Immediate dominated*: If a branch calculates the target address based on a full-address size immediate, we can assume that the probability of execution depends on the memory occupancy of the process, because the immediate is just another random number generated by the application of the mask to the attack code.

We use this classification in empirical studies of random code execution (Section 4.2). These experiments provide evidence that most processes terminate quickly when random code sequences are inserted. We then describe a theoretical model for the execution of random IA32 and PowerPC instructions (Section 5), which allows us to validate the experiments and provides a framework for future analysis of other architectures.

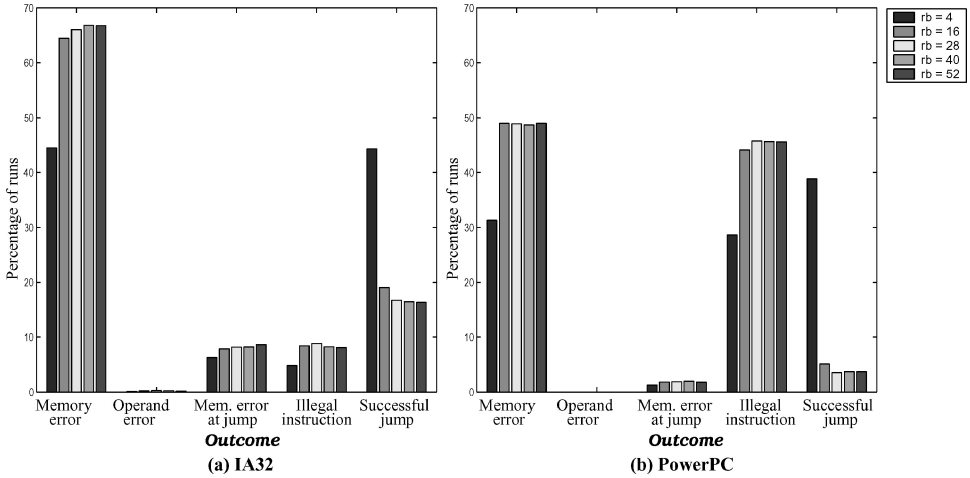

Fig. 2. Executing random blocks on native processors. The plots show the distribution of runs by type of outcome for (a) IA32 and (b) Power PC. Each color corresponds to a different random block size (*rb*): 4, 16, 28, 40, and 52 bytes. The filler is set such that the total process density is 5% of the possible 2^{32} address space. The experiment was run under the Linux operating system.

4.2 Empirical Testing

We performed two kinds of experiments: (1) execution of random blocks of bytes on native processors, and (2) execution of real attacks in RISE on IA32.

4.2.1 *Executing Blocks of Random Code.* We wrote a simple C program that executes blocks of random bytes. The block of random bytes simulates a randomized exploit running under RISE. We then tested the program for different block sizes (the "exploit") and different degrees of process space occupancy. The program allocates a prespecified amount of memory (determined by the *filler size* parameter) and fills it with the machine code for no operation (NOP). The block of random bytes is positioned in the middle of the filler memory.

Figure 2 depicts the observed frequency of the events defined in Section 4.1. There is a preponderance of memory access errors in both architectures, although the less dense PowerPC has an almost equal frequency of illegal instructions. Illegal instructions occur infrequently in the IA32 case. In both architectures, about one-third of legal branch instructions fail because of an invalid memory address, and two-thirds manage to execute the branch. Conditional branches form the majority of branch instructions in most architectures, and these branches have a high probability of executing because of their very short relative offsets.

Because execution probabilities could be affected by the memory occupancy of the process, we tested different process memory sizes. The process sizes used are expressed as fractions of the total possible 2^{32} address space (Table IV).

Each execution takes place inside GDB (the GNU debugger), single stepping until either a signal occurs or more than 100 instructions have been executed. We collect information about type of instruction, addresses, and types of signals during the run. We ran this scenario with 10,000 different seeds, five random

Table IV. Process Memory Densities (Relative to Process Size)

Process Memory Density (as a fraction of 2^{32} bytes)	0.0002956	0.0036093	0.0102365	0.0234910	0.0500000

Values are expressed as fractions of the total possible 2^{32} address space. They are based on observed process memory used in two busy IA32 Linux systems over a period of two days.

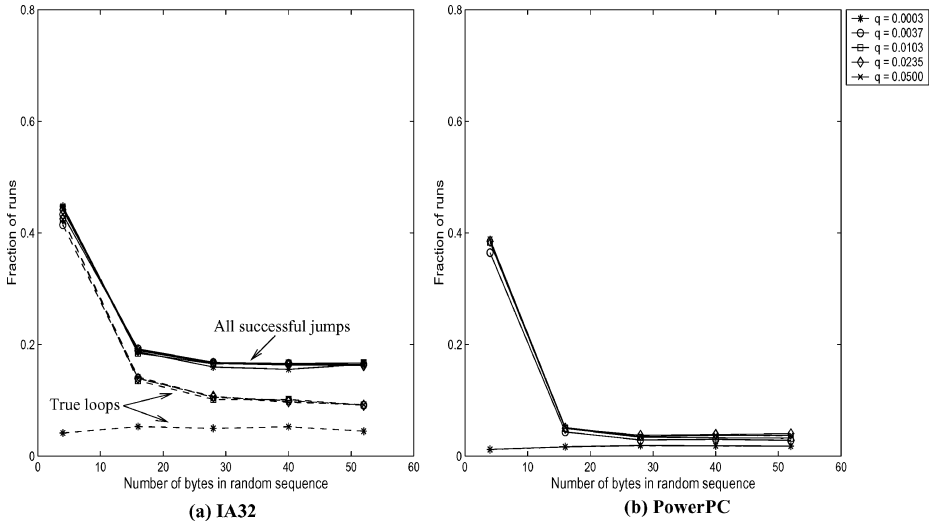

Fig. 3. Probability that random code escapes when executed for different block sizes (the x-axis) for (a) IA32 and (b) Power PC. Block size is the length of the sequence of random bytes inserted into the process. Each set of connected points represents a different memory density (q). Solid lines represent the fraction of runs that escaped under our definition of escape, and dotted lines show the fraction of "true" escaped executions (those that did not fail after escaping from the exploit area).

block sizes (4, 8, 24, 40, and 56 bytes), and five total process densities (see Table IV), both for the PowerPC and the IA32.

Figure 3 plots the fraction of runs that *escaped* according to our definition of escape (given in Section 4.1) for different memory densities. An execution was counted as an escape if a jump was executed and did not fail immediately (that is, it jumped to an executable section of the code). In addition, it shows the proportion of escapes that did not crash within a few bytes of the exploit area ("true" escapes: for example when the execution is trapped into an infinite loop). Escapes that continued executing for more than 100 instructions were terminated. The figure shows that for realistic block sizes (over 45 bytes), the proportion of true escapes is under 10% (IA32). In the Power PC case, although the fraction of escaped runs is smaller, most of the escapes do not fail afterwards, so the curves overlap.

A second observation (not shown) is that memory density has a negligible effect on the probability of escape, even though we created an environment that maximizes successful escapes. This is likely because the process sizes are still relatively small compared to the total address space and because only a minority of memory accesses are affected by this density (those that are immediate dominated).

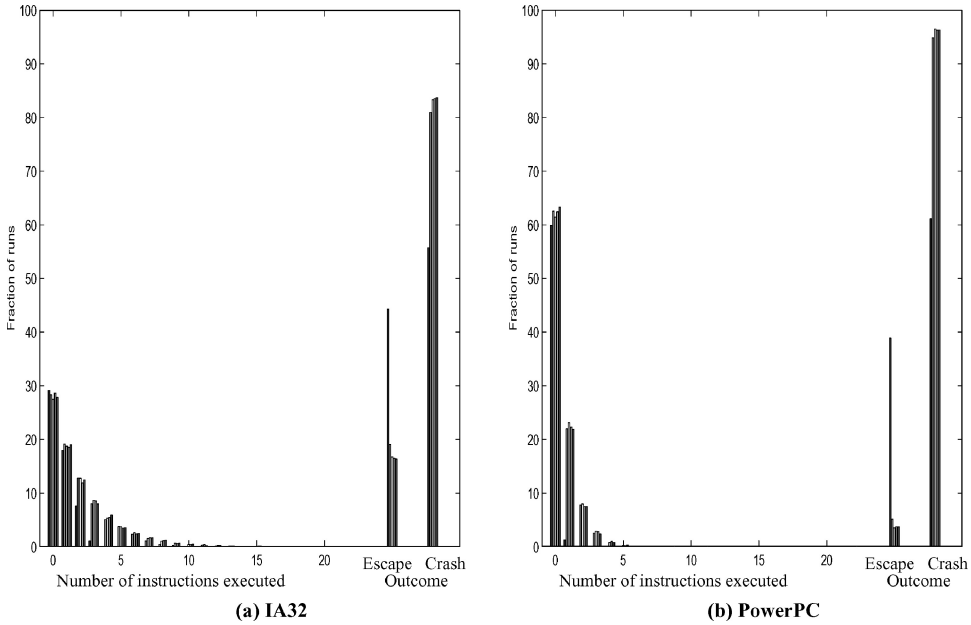

Fig. 4. Proportion of runs that fail after exactly n instructions, with memory density 0.05, for (a) IA32 and (b) PowerPC. On the right, the proportion of escaped versus crashed runs is presented for comparison. Each instruction length bar is composed by five sub-bars, one for each random block (simulated attack) sizes 4, 16, 28, 40, and 52 bytes, left to right.

Figure 4 shows the proportion of failed runs that die after executing exactly n instructions. On the right side of the graph, the proportion of escaped versus failed runs is shown for comparison. Each instruction length bar comprises five subbars, one for each simulated attack size. We plot them all to show that the size of the attack has almost no effect on the number of instructions executed, except for very small sizes. On the IA32, more than 90% of all failed runs died after executing at most 6 instructions and in no case did the execution continue for more than 23 instructions. The effect is even more dramatic on the Power PC, where 90% of all failed runs executed for fewer than 3 instructions, and the longest failed run executed only 10 instructions.

4.2.2 *Executing Real Attacks under RISE.* We ran several vulnerable applications under RISE and attacked them repeatedly over the network, measuring how long it took them to fail. We also tested the two synthetic attacks from Fayolle and Glaume [2002]. In this case the attack and the exploit are in the same program, so we ran them in RISE for 10,000 times each, collecting output from RISE. Table V summarizes the results of these experiments. The real attacks fail within an average of two to three instructions (column 4). Column 3 shows how many attack instances we ran (each with a different random seed for masking) to compute the average. As column 5 shows, most attack instances crashed instead of escaping. The synthetic attacks averaged just under two instructions before process failure. No execution of any of the attacks was able to spawn a shell.

Table V. Survival Time in Executed Instructions for Attack Codes in Real-Applications Running under RISE

Attack Name	Application	No. of Attacks	Avg. no. of Insns.	Crashed Before Escape (%)
Named NXT resource record overflow	Bind 8.2.1-7	101	2.24	85.14
rpc.statd format string	nfs-utils 0.1.6-2	102	2.06	85.29
Samba trans2 exploit	smbd 2.2.1a	81	3.13	73.00
Synthetic heap exploit	N/A	10,131	1.98	93.93
Synthetic stack exploit	N/A	10,017	1.98	93.30

Column 4 gives the average number of instructions executed before failure (for instances that did not "escape").
Column 5 summarizes the percentage of runs crashing (instead of "escaping").

Within the RISE approach, one could avoid the problem of accidentally viable code by mapping to a larger instruction set. The size could be tuned to reflect the desired percentage of incorrect unscramblings that will likely lead immediately to an illegal instruction.

5. RISE SAFETY: THEORETICAL ANALYSIS

This section develops theoretical estimates of RISE safety and compares them with the experiments reported in the previous section. A theoretical analysis is important for several reasons. Diversified code techniques of various sorts and at various levels are likely to become more common. We need to understand exactly how much protection they confer. In addition, it will be helpful to predict the effect of code diversity on new architectures before they are built. For example, analysis allows us to predict how much increase in safety could be achieved by expanding the size of the instruction space by a fixed amount.

In the case of a variable-size instruction set, such as the IA32, we compute the aggregate probabilities using a Markov chain. In the case of a uniform-length instruction set, such as the PowerPC, we can compute the probabilities directly.

5.1 IA32 Instruction Set

For the IA32 instruction set, which is a CISC architecture, we use the published instruction set specification [Intel Corporation 2004] to build a Markov chain used to calculate the escape probability of a sequence of m random bytes (with byte-length $b = 8$ bits). Our analysis is based on the graph of event categories shown in Figure 1, but it is specialized to include the byte-to-byte symbol recognition transitions. A transition is defined as the reading of a byte by the processor, and the states describe either specific positions within instructions or exceptions. Appendix A provides the specifics of this particular Markov chain encoding.

Apart from the complexity of encoding the large and diverse IA32 instruction set, the major difficulty in the model is the decision of what to do when a symbol

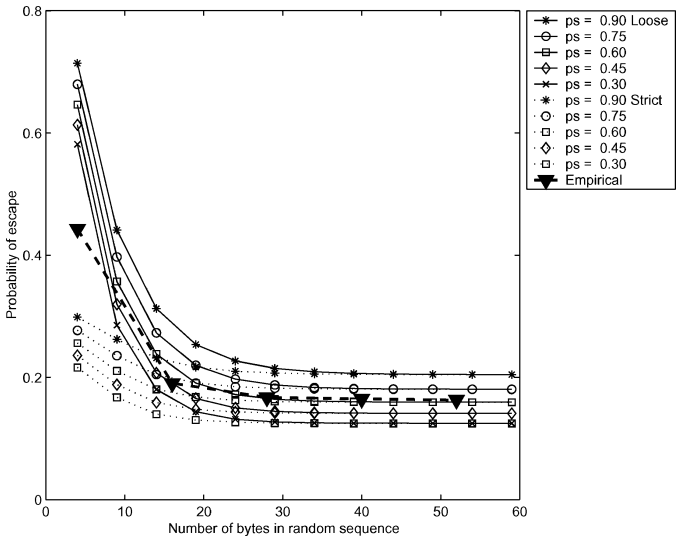

Fig. 5. Theoretical analysis of IA32 escape probability: The x-axis is the number of bytes in the random sequence, and the y-axis is the probability of escaping from a random string of m bytes. Each connected set of plotted points corresponds to one assumed probability of successfully executing a process-state-dominated memory access (p_s), with either strict or loose criterion of escape. The memory density is fixed at 0.05. For comparison with empirical data, the dashed line with triangles marks the observed average frequency of successful jumps (data taken from Figure 3 for the IA32 and memory density $q = 0.05$).

crosses the boundary of the exploit area. It is conceivable that the result of the interpretation is still ambiguous at the byte that crosses the border. However, the model needs to decide what happens to the execution at that moment. We model this situation using both extremes: A *loose escape* declares the execution as an escape if the bytes interpreted up to the crossing point have not caused a crash; a *strict escape* declares that the execution of the partial instruction ends in crash. A characterization of the states in terms of the Markov chain is presented in Appendix A.

Figure 5 shows the probability of escape as a function of increasing sequence length for both loose and strict criteria of escape for a fixed memory density (0.05), and for different probabilities of a process state-dominated memory access to execute. The plot reveals several interesting phenomena.

First, the plots converge to a steady state quickly—in less than 20 bytes. This is consistent with the empirical data. Notably, the probability of escape converges to a nonzero value. This means that independently of exploit or process size, there will always be a nonzero probability that a sequence of random code will escape.

A second observation revealed by the plot is the relatively small difference between our loose and strict criteria for escape. The main difference between both cases is how to interpret the last instruction in the sequence if the string has not crashed before the exploit border. Not surprisingly, as sequences get longer, the probability of reaching the last symbol diminishes, so the overall effect of an ambiguous last instruction in those few cases is respectively smaller.

Table VI. Partition of Symbols into Disjoint Sets

Set Name	Type of Instructions in Set
U	Undefined instructions.
P	Privileged instructions.
B_{SR}	Small offset, relative branch
L_D	Legal instructions with no memory access and no branching. All branches require memory access, so L_D only contains linear instructions.
L_{MI}	Legal no-branch instructions with immediate-dominated memory access.
B_{MI}	Legal branch instructions with immediate-dominated memory-access.
L_{MP}	Legal no-branch instructions with process-state dominated memory-access.
B_{MP}	Legal branch instructions with process-state dominated memory access.

A third observation (data not shown in the figure) is that for different memory densities, the escape curves are nearly identical. This means that memory size has almost no effect on the probability of escape at typical process memory occupancies. In part, this reflects the fact that most jumps use process-state-dominated memory accesses. In particular, immediate-dominated memory accesses constitute a very small proportion of the instructions that use memory (only 4 out of more than 20 types of jumps).

The fourth observation concerns the fact that the first data point in the empirical run (block size of 4 bytes) differs markedly from all the strict and loose predicted curves. Both criteria are extreme cases and the observed behavior is in fact bounded by them. The divergence is most noticeable during the first 10 bytes, as most IA32 instructions have a length between 4 and 10 bytes. As noted before, the curves for loose and strict converge rapidly as the effect of the last instruction becomes less important, and so we see a much closer fit with the predicted behavior after 10 bytes, as the bounds become tighter.

The final observation is that the parameter p_s varies less than expected. We were expecting that the empirical data would have an ever-increasing negative slope, given that in principle the entropy of the process would increase as more instructions were executed. Instead, we get a close fit with $p_s = 0.6$ after the first 20 bytes. This supports our approximation to the probability of execution for process-state dominated instructions, as a constant that can be determined with system profiling.

5.2 Uniform-Length Instruction Set Model

The uniform-length instruction set is simpler to analyze because it does not require conditional probabilities on instruction length. Therefore, we can estimate the probabilities directly without resorting to a Markov chain. Our analysis generalizes to any RISC instruction set, but we use the PowerPC [IBM 2003] as an example.

Let all instructions be of length b bits (usually $b = 32$). We calculate the probability of escape from a random string of m symbols $r = r_1 \ldots r_m$, each of length b bits (assumed to be drawn from a uniform distribution of 2^b possible symbols). We can partition all possible symbols in disjoint sets with different execution characteristics. Table VI lists the partition we chose to use. Figure 7 in Appendix B illustrates the partition in terms of the classification of events

given in Section 4.1. $S = U \cup P \cup B_{SR} \cup L_D \cup L_{MI} \cup B_{MI} \cup L_{MP} \cup B_{MP}$ is the set of all possible symbols that can be formed with b bits. $|S| = 2^b$. The probability that a symbol s belongs to any given set I (where I can be any one of U, P, B_{SR}, L_D, L_{MI}, B_{MI}, L_{MP} or B_{MP}) is given by $P\{s \in I\} = P(I) = \frac{|I|}{2^b}$.

If there are a bits for addressing (and consequently the size of the address space is 2^a); E_I is the event that a symbol belonging to set I executes; M_t is the total memory space allocated to the process; M_e is the total executable memory of the process; and p_s is the probability that a memory access dominated by the processor state succeeds, then the probabilities of successful execution for instructions in each set are

For illegal and privileged opcodes, $P(E_U) = P(E_P) = 0$.
For the remaining legal opcodes, $P(E_{L_D}) = P(E_{B_{SR}}) = 1$; $P(E_{L_{MI}}) = \frac{M_t}{2^a}$; $P(E_{B_{MI}}) = \frac{M_e}{2^a}$; $P(E_{L_{MP}}) = p_s$ and $P(E_{B_{MP}}) = p_s$.

We are interested in the probability of a successful branch (escape) out of a sequence of n random bytes. Let X_n denote the event that an execution escapes at *exactly* symbol n. This event requires that $n-1$ instructions execute without branching and that the nth instruction branches successfully. In consequence, $P(X_n) = (P(L))^{n-1}P(E)$, where $P(L) = P(L_D) + P(L_{MI})P(E_{L_{MI}}) + P(L_{MP})$ is the probability that a symbol executes a successful linear instruction, and $P(E) = P(B_{MI})P(E_{B_{MI}}) + P(B_{MP}) + P(B_{SR})$ is the probability that a symbol executes a valid branch.

If $X_n^\star$ is the event that the execution of a random string $r = r_1 \cdots r_n$ escapes, its probability $P(X_n^\star)$ is given by (Appendix B:

$$P(X_n^\star) = P(E)\frac{1 - P(L)^{n+1}}{1 - P(L)} + P(L)^n$$

$P(X_n^\star)$ is plotted in Figure 6 for different values of p_s, increasing random code sizes and a given memory density (0.05 as in the IA32 case). The comparable data points from our experiments are shown for comparison. We did not plot results for different memory densities because the difference among the curves is negligible. The figure shows that the theoretical analysis agrees with our experimental results. The parameters were calculated from the published documentation of the PowerPC instruction set [IBM 2003], for the 32-bit case: $b = 32$, $a = 32$, $P(L_D) \approx 0.25$, $P(L_{MI}) = 0$, $P(L_{MP}) \approx 0.375$, $P(B_{MI}) \approx 0.015$, $P(B_{MP}) \approx 0.030$, $P(B_{SR}) \approx 0.008$.

It can be seen that the probability of escape converges to a nonzero value. For a uniform-length instruction set, this value can be calculated as

$$\lim_{n\to\infty} P(X_n^\star) = \frac{P(E)}{1 - P(L)}.$$

The limit value of $P(X_n^\star)$ is the lower bound on the probability of a sequence of length n escaping. It is independent of n, so larger exploit sizes are no more likely to fail than smaller ones in the long run. It is larger than 0 for any architecture in which the probability of successful execution of a jump to a random location is larger than 0.

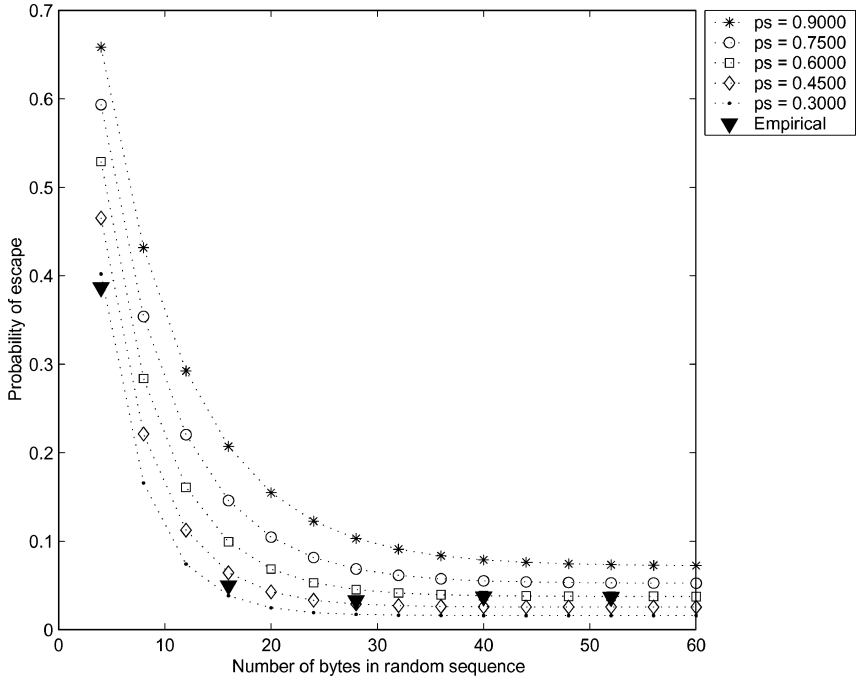

Fig. 6. Theoretical probability of escape for a random string of n symbols. Each curve plots a different probability of executing a process-state-determined memory access (p_s) for the PowerPC uniform-length instruction set. Process memory occupancy is fixed at 0.05. The large triangles are the measured data points for the given memory occupancy (data taken from Figure 3 for the PowerPC and memory density $q = 0.05$), and the dotted lines are the predicted probabilities of escape.

6. RELATED WORK

Our randomization technique is an example of *automated diversity*, an idea that has long been used in software engineering to improve fault tolerance [Avizienis 1995; Avizienis and Chen 1977; Randell 1975] and more recently has been proposed as a method for improving security [Cohen 1993; Forrest et al. 1997; Pu et al. 1996]. The RISE approach was introduced in Barrantes et al. [2003], and an approach similar to RISE was proposed in Kc et al. [2003].

Many other approaches have been developed for protecting programs against particular methods of code injection, including: static code analysis [Dor et al. 2003; Larochelle and Evans 2001; Wagner et al. 2000] and runtime checks, using either static code transformations [Avijit et al. 2004; Baratloo et al. 2000; Chiueh and Hsu 2001; Cowan et al. 1998, 2001; Etoh and Yoda 2000, 2001; Jones and Kelly 1997; Lhee and Chapin 2002; Nebenzahl and Wool 2004; Prasad and Chiueh 2003; Ruwase and Lam 2004; Tsai and Singh 2001; Vendicator 2000; Xu et al. 2002], dynamic instrumentation [Baratloo et al. 2000; Kiriansky et al. 2002], or hybrid schemes [Jim et al. 2002; Necula et al. 2002]. In addition, some methods focus on protecting an entire system rather than a particular program, resulting in defense mechanisms at the operating system level and hardware support [Milenković et al. 2004; PaX Team 2003; Xu et al. 2002]. Instruction-set

randomization is also related to hardware code encryption methods explored in Kuhn [1997] and those proposed for TCPA/TCG [TCPA 2004].

6.1 Automated Diversity

Diversity in software engineering is quite different from diversity for security. In software engineering, the basic idea is to generate multiple independent solutions to a problem (e.g., multiple versions of a software program) with the hope that they will fail independently, thus greatly improving the chances that some solution out of the collection will perform correctly in every circumstance. The different solutions may or may not be produced manually, and the number of solutions is typically quite small, around 10.

Diversity in security is introduced for a different reason. Here, the goal is to reduce the risk of widely replicated attacks, by forcing the attacker to redesign the attack each time it is applied. For example, in the case of a buffer overflow attack, the goal is to force the attacker to rewrite the attack code for each new computer that is attacked. Typically, the number of different diverse solutions is very high, potentially equal to the total number of program copies for any given program. Manual methods are thus infeasible, and the diversity must be produced automatically.

Cowan et al. [2000] introduced a classification of diversity methods applied to security (called "security adaptations") which classifies diversifications based on what is being adapted—either the interface or the implementation. Interface diversity modifies code layout or access controls to interfaces, without changing the underlying implementation to which the interface gives access. Implementation diversity, on the other hand, modifies the underlying implementation of some portion of the system to make it resistant to attacks. RISE can be viewed as a form of interface diversity at the machine code level.

In 1997, Forrest et al. presented a general view of the possibilities of diversity for security [Forrest et al. 1997], introducing the idea of deliberately diversifying data and code layouts. They used the example of randomly padding stack frames to make exact return address locations less predictable, and thus more difficult for an attacker to locate. Developers of buffer overflow attacks have developed a variety of workarounds—such as "ramps" and "landing zones" of no-ops and multiple return addresses. Automated diversity via random stack padding coerces an attacker to use such techniques; it also requires larger attack codes in proportion to the size range of random padding employed.

Other work in automated diversity for security has also experimented with diversifying data layouts [Cohen 1993; Pu et al. 1996], as well as system calls [Chew and Song 2002], and file systems [Cowan et al. 2000]. In addition, several projects address the code-injection threat model directly, and we describe those projects briefly.

Chew and Song [2002] proposed a method that combines kernel and loader modification on the system level with binary rewriting at the process level to provide system call number randomization, random stack relocation, and randomization of standard library calls. This work has not been completely evaluated to our knowledge.

Address space layout randomization (ASLR) [PaX Team 2003] and transparent runtime randomization (TRR) [Xu et al. 2003] randomize the positions of the stack, shared libraries, and heap. The main difference between the two is the implementation level. ASLR is implemented in the kernel, while TRR modifies the loader program. Consequently, TRR is more oriented to the end user.

Bhatkar et al. [2003] describe a method that randomizes the addresses of data structures internal to the process, in addition to the base address of the main segments. Internal data and code blocks are permuted inside the segments and the guessing range is increased by introducing random gaps between objects. The current implementation instruments object files and ELF binaries to carry out the required randomizations. No access to the source code is necessary, but this makes the transformations extremely conservative. This technique nicely complements that of RISE, and the two could be used together to provide protection against both code injection and return-into-libc attacks simultaneously.

PointGuard [Cowan et al. 2003] uses automated randomization of pointers in the code and is implemented by instrumenting the intermediate code (AST in GCC).

The automated diversity project that is closest to RISE is the system described in Kc et al. [2003], which also randomizes machine code. There are several interesting points of comparison with RISE, and we describe two of them: (1) persystem (whole image) versus perprocess randomization; (2) Bochs [Butler 2004] versus Valgrind as emulator. First, in the Kc et al. implementation, a single key is used to randomize the image, all the libraries, and any applications that need to be accessed in the image. The system later boots from this image. This has the advantage that in theory, kernel code could be randomized using their method although most code-injection attacks target application code. A drawback of this approach lies in its key management. There is a single key for all applications in the image, and the key cannot be changed during the lifetime of the image. Key guessing is a real possibility in this situation, because the attacker would be likely to know the cleartext of the image. However, the Kc et al. system is more compact because there is only one copy of the libraries. On the other hand, if the key is guessed for any one application or library, then all the rest are vulnerable. Second, the implementations differ in their choice of emulator. Because Bochs is a pure interpreter it incurs a significant performance penalty, while emulators such as Valgrind can potentially achieve close-to-native efficiency through the use of optimized and cached code fragments.

A randomization of the SQL language was proposed in Boyd and Keromytis [2004]. This technique is essentially the same one used in the Perl randomizer [Kc et al. 2003], with a random string added to query keywords. It is implemented through a proxy application on the server side. In principle, there could be one server proxy per database connection, thus allowing more key diversity. The performance impact is minimal, although key capture is theoretically possible in a networked environment.

6.2 Other Defenses Against Code Injection

Other defenses against code injection (sometimes called "restriction methods") can be divided into methods at the program and at the system level. In turn, approaches at the program level comprise static code analysis and runtime code instrumentation or surveillance. System level solutions can be implemented in the operating system or directly through hardware modifications. Of these, we focus on the methods most relevant to RISE.

6.2.1 *Program-Level Defenses Against Code Injection.* Program-level approaches can be seen as defense-in-depth, beginning with suggestions for good coding practices and/or use of type-safe languages, continuing with automated analysis of source code, and finally reaching static or dynamic modification of code to monitor the process progress and detect security violations. Comparative studies on program-level defenses against buffer overflows have been presented by Fayolle and Glaume [2002], Wilander and Kamkar [2003], and Simon [2001]. Several relevant defenses are briefly discussed below.

The StackGuard system [Cowan et al. 1998] modifies GCC to interpose a a canary word before the return address, the value of which is checked before the function returns. An attempt to overwrite the return address via linear stack smashing will change the canary value and thus be detected.

StackShield [Vendicator 2000], RAD [Chiueh and Hsu 2001], install-time vaccination [Nebenzahl and Wool 2004], and binary rewriting[Prasad and Chiueh 2003] all use instrumentations to store a copy of the function return address off the stack and check against it before returning to detect an overwrite. Another variant, Propolice [Etoh and Yoda 2000, 2001] uses a combination of a canary word and frame data relocation to avoid sensible data overwriting. Split control and data stack [Xu et al. 2002] divides the stack in a control stack for return addresses and a data stack for all other stack-allocated variables.

FormatGuard [Cowan et al. 2001] used the C preprocessor (CPP) to add parameter-counting to printf-like C functions and defend programs against format print vulnerabilities. This implementation was not comprehensive even against this particular type of attacks.

A slightly different approach uses wrappers around standard library functions, which have proven to be a continuous source of vulnerabilities. Libsafe [Baratloo et al. 2000; Tsai and Singh 2001], TIED, and LibsafePlus [Avijit et al. 2004], and the type-assisted bounds checker proposed by Lhee and Chapin [2002] intercept library calls and attempt to ensure that their manipulation of user memory is safe.

An additional group of techniques depends on runtime bounds checking of memory objects, such as the Kelly and Jones bound checker [Jones and Kelly 1997] and the recent C range error detector (CRED) [Ruwase and Lam 2004]. Their heuristics differ in the way of determining if a reference is still legal. Both can generate false positives, although CRED is less computationally expensive.

The common theme in all these techniques is that they are specific defenses, targeting specific points of entry for the injected code (stack, buffers, format

functions, and so on). Therefore, they cannot prevent an injection arriving from a different source or an undiscovered vulnerability type. RISE, on the other hand, is a generic defense that is independent of the method by which binary code is injected.

There is also a collection of dynamic defense methods which do not require access to the original sources or binaries. They operate directly on the process in memory, either by inserting instrumentation as extra code (during the load process or as a library) or by taking complete control as in the case of native-to-native emulators.

Libverify [Baratloo et al. 2000] saves a copy of the return address to compare at the function end, so it is a predecessor to install-time vaccination [Nebenzahl and Wool 2004] and binary rewriting [Prasad and Chiueh 2003], with the difference that it is implemented as a library that performs the rewrite dynamically, so the binaries on disk do not require modification.

Code shepherding [Kiriansky et al. 2002] is a comprehensive, policy-based restriction defense implemented over a binary-to-binary optimizing emulator. The policies concern client code control transfers that are intrinsically detected during the interpretation process. Two of those types of policies are relevant to the RISE approach.

Code origin policies grant differential access based on the source of the code. When it is possible to establish if the instruction to be executed came from a disk binary (modified or unmodified) or from dynamically generated code (original or modified after generation), policy decisions can be made based on that origin information. In our model, we are implicitly implementing a code origin policy, in that only unmodified code from disk is allowed to execute. An advantage of the RISE approach is that the origin check cannot be avoided—only properly sourced code is mapped into the private instruction set so it executes successfully. Currently, the only exception we have to the disk origin policy is for the code deposited in the stack by signals. RISE inherits its signal manipulation from Valgrind [Nethercote and Seward 2003]. More specifically, all client signals are intercepted and treated as special cases. Code left on the stack is executed separately from the regular client code fetch cycle so it is not affected by the scrambling. This naturally resembles PaX's special handling of signals, where code left on the stack is separately emulated.

Also relevant are restricted control transfers in which a transfer is allowed or disallowed according to its source, destination, and type. Although we use a restricted version of this policy to allow signal code on the stack, in most other cases we rely on the RISE language barrier to ensure that injected code will fail.

6.2.2 *System-Level Defenses Against Code Injection.* System level restriction techniques can be applied in the operating system, hardware, or both. We briefly review some of the most important system-level defenses.

The nonexecutable stack and heap as implemented in the PAGEEXEC feature of PaX [PaX Team 2003] is hardware assisted. It divides allocation into data and code TLBs and intercepts all page-fault handlers into the code TLB. As with any hardware-assisted technique, it requires changes to the kernel.

RISE is functionally similar to these techniques, sharing the ability to randomize ordinary executable files with no special compilation requirements. Our approach differs, however, from nonexecutable stacks and heaps in important ways. First, it does not rely on special hardware support (although RISE pays a performance penalty for its hardware independence). Second, although a system administrator can choose whether to disable certain PaX features on a perprocess basis, RISE can be used by an end-user to protect user-level processes without any modification to the overall system.

A third difference between PaX and RISE is in how they handle applications that emit code dynamically. In PaX, the process-emitting code requires having the PAGEEXEC feature disabled (at least), so the process remains vulnerable to injected code. If such a process intended to use RISE, it could modify the code-emitting procedures to use an interface provided by RISE, and derived from Valgrind's interface for Valgrind-aware applications. The interface uses a validation scheme based on the original randomization of code from disk. In a pure language randomization, a process-emitting dynamic code would have to do so in the particular language being used at that moment. In our approximation, the process using the interface scrambles the new code before execution. The interface, a RISE function, considers the fragment of code as a new library, and randomizes it accordingly. In contrast to nonexecutable stack/heap, this does not make the area where the new code is stored any more vulnerable, as code injected in this area will still be expressed in nonrandomized code and will not be able to execute except as random bytes.

Some other points of comparison between RISE and PaX include

(1) *Resistance to return-into-libc*: Both RISE and PaX PAGEEXEC features are susceptible to return-into-libc attacks when implemented as an isolated feature. RISE is vulnerable to return-into-libc attacks without an internal data structure randomization, and data structure randomization is vulnerable to injected code without the code randomization. Similarly, as the PaX Team notes, LIBEEXEC is vulnerable to return-into-libc without ASLR (automatic stack and library randomization), and ASLR is vulnerable to injected code without PAGEEXEC [PaX Team 2003]. In both cases, the introduction of the data structure randomization (at each corresponding granularity level) makes return-into-libc attacks extremely unlikely.
(2) *Signal code on the stack*: Both PaX and RISE support signal code on the stack. They both treat it as a special case. RISE in particular is able to detect signal code as it intercepts all signals directed to the emulated process and examines the stack before passing control to the process.
(3) *C trampolines*: PaX detects trampolines by their specific code pattern and executes them by emulation. The current RISE implementation does not support this, although it would not be difficult to add it.

StackGhost [Frantzen and Shuey 2001] is a hardware-assisted defense implemented in OpenBSD for the Sparc architecture. The return address of functions is stored in registers instead of the stack, and for a large number of nested

calls StackGhost protects the overflowed return addresses through write protection or encryption.

Milenković et al. [2004] propose an alternative architecture where linear blocks of instructions are signed on the last basic block (equivalent to a line of cache). The signatures are calculated at compilation time and loaded with the process into a protected architectural structure. Static libraries are compiled into a single executable with a program, and dynamic libraries have their own signature file loaded when the library is loaded. Programs are stored unmodified, but their signature files should be stored with strong cryptographic protection. Given that the signatures are calculated once, at compile time, if the signature files are broken, the program is vulnerable.

Xu et al. [2002] propose using a secure return address stack (SRAS) that uses the redundant copy of the return address maintained by the processor's fetch mechanism to validate the return address on the stack.

6.3 Hardware Encryption

Because RISE uses runtime code scrambling to improve security, it resembles some hardware-based code encryption schemes. Hardware components to allow decryption of code and/or data on-the-fly have been proposed since the late 1970s [Best 1979, 1980] and implemented as microcontrollers for custom systems (for example, the DS5002FP microcontroller [Dallas Semiconductor 1999]). The two main objectives of these cryptoprocessors are to protect code from piracy and data from in-chip eavesdropping. An early proposal for the use of hardware encryption in general-purpose systems was presented by Kuhn [1997] for a very high threat level where encryption and decryption were performed at the level of cache lines. This proposal adhered to the model of protecting licensed software from users, and not users from intruders, so there was no analysis of shared libraries or how to encrypt (if desired) existing open applications. A more extensive proposal was included as part of TCPA/TCG [TCPA 2004]. Although the published TCPA/TCG specifications provide for encrypted code in memory, which is decrypted on the fly, TCPA/TCG is designed as a much larger authentication and verification scheme and has raised controversies about digital rights management (DRM) and end-users' losing control of their systems [Anderson 2003; Arbaugh 2002]. RISE contains none of the machinery found in TCPA/TCG for supporting DRM. On the contrary, RISE is designed to maintain control locally to protect the user from injected code.

7. DISCUSSION

The preceding sections describe a prototype implementation of the RISE approach and evaluate its effectiveness at disrupting attacks. In this section, we address some larger questions about RISE.

7.1 Performance Issues

Although Valgrind has some limitations, discussed in Section 2, we are optimistic that improved designs and implementations of "randomized machines" would improve performance and reduce resource requirements, potentially

expanding the range of attacks the approach can mitigate. We have also observed that even in its current version, the performance RISE offers could be acceptable if the processes are I/O bound and/or use the network extensively.

In the current implementation, RISE safety is somewhat limited by the dense packing of legal IA32 instructions in the space of all possible byte patterns. A random scrambling of bits is likely to produce a different legal instruction. Doubling the size of the instruction encoding would enormously reduce the risk of a processor's successfully executing a long enough sequence of unscrambled instructions to do damage. Although our preliminary analysis shows that this risk is low even with the current implementation, we believe that emerging soft-hardware architectures such as Crusoe [Klaiber 2000] will make it possible to reduce the risk even further.

7.2 Is RISE Secure?

A valid concern when evaluating RISE's security is its susceptibility to key discovery, as an attacker with the appropriate scrambling information could inject scrambled code that will be accepted by the emulator. We believe that RISE is highly resistant to this class of attack.

RISE is resilient against brute force attacks because the attacker's work is exponential in the shortest code sequence that will make an externally detectable difference if it is unscrambled properly. We can be optimistic because most IA32 attack codes are at least dozens of bytes long, but if a software flaw existed that was exploitable with, say, a single 1-byte opcode, then RISE would be vulnerable, although the process of guessing even a 1-byte representation would cause system crashes easily detectable by an administrator.

An alternative path for an attacker is to try to inject arbitrary address ranges of the process into the network, and recover the key from the downloaded information. The download could be part of the key itself (stored in the process address space), scrambled code, or unscrambled data. Unscrambled data does not give the attacker any information about the key. Even if the attacker could obtain scrambled code or pieces of the key (they are equivalent because we can assume that the attacker has knowledge of the program binary), using the stolen key piece might not be feasible. If the key is created eagerly, with a key for every possible address in the program, past or future, then the attacker would still need to know where the attack code is going to be written in process space to be able to use that information. However, in our implementation, where keys are created lazily for code loaded from disk, the key for the addresses targeted by the attack might not exist, and therefore might not be discoverable. The keys that do exist are for addresses that are usually not used in code injection attacks because they are write protected. In summary, it would be extremely difficult to discover or use a particular encoding during the lifetime of a process.

Another potential vulnerability is RISE itself. We believe that RISE would be difficult to attack for several reasons. First, we are using a network-based threat model (attack code arrives over a network) and RISE does not perform network reads. In fact it does not read any input at all after processing the run arguments. Injecting an attack through a flawed RISE read is thus impossible.

Second, if an attack arises inside a vulnerable application and the attacker is aware that the application is being run under RISE, the vulnerable points are the code cache and RISE's stack, as an attacker could deposit code and wait until RISE proceeds to execute something from these locations. Although RISE's code is not randomized because it has to run natively, the entire area is write protected, so it is not a candidate for injection. The cache is read-only during the time that code blocks are executed, which is precisely when this hypothetical attack would be launched, so injecting into the cache is infeasible.

Another possibility is a *jump-into-RISE* attack. We consider three ways in which this might happen:[3]

(1) The injected address of RISE code is in the client execution path cache.
(2) The injected address of RISE code is in the execution path of RISE itself.
(3) The injected address of RISE code is in a code fragment in the cache.

In case 1, the code from RISE will be interpreted. However, RISE only allows certain self-functions to be called from client code, so everything else will fail. Even for those limited cases, RISE checks the call origin, disallowing any attempt to modify its own structures.

For case 2, the attacker would need to inject the address into a RISE data area in RISE's stack or in an executable area. The executable area is covered by case 3. For RISE's data and stack areas we have introduced additional randomizations. The most immediate threat is the stack, so we randomize its start address. For other data structures, the location could be randomized using the techniques proposed in Bhatkar et al. [2003], although this is unimplemented in the current prototype. Such a randomization would make it difficult for the attacker to guess its location correctly. An alternative, although much more expensive, solution would be to monitor all writes and disallow modifications from client code and certain emulator areas.

It is worth noting that this form of attack (targeting emulator data structures) would require executing several commands without executing a single machine language instruction. Although such attacks are theoretically possible via chained system calls with correct arguments, and simple (local) attacks have been shown to work [Nergal 2001], these are not a common technique [Wilander and Kamkar 2003]. In the next version of RISE, we plan to include full data structure address randomization, which would make these rare attacks extremely difficult to execute.

Case 3 is not easily achieved because fragments are write protected. However, an attacker could conceivably execute an *mprotect* call to change writing rights and then write the correct address. In such a case, the attack would execute. This is a threat for applications running over emulators, as it undermines all other security policies [Kiriansky et al. 2002]. In the current RISE implementation, we borrow the solution used in Kiriansky et al. [2002], monitoring all calls to the *mprotect* system call by checking their source and destination and not allowing executions that violate the protection policy.

[3]We rely on the fact that RISE itself does not receive any external input once it is running.

7.3 Code/Data Boundaries

An essential requirement for using RISE for improving security is that the distinction between code and data must be carefully maintained. The discovery that code and data can be systematically interchanged was a key advance in early computer design, and this dual interpretation of bits as both numbers and commands is inherent to programmable computing. However, all that flexibility and power turn into security risks if we cannot control how and when data become interpreted as code. Code-injection attacks provide a compelling example, as the easiest way to inject code into a binary is by disguising it as data, for example, as inputs to functions in a victim program.

Fortunately, code and data are typically used in very different ways, so advances in computer architecture intended solely to improve performance, such as separate instruction caches and data caches, also have helped to enforce good hygiene in distinguishing machine code from data, helping make the RISE approach feasible. At the same time, of course, the rise of mobile code, such as Javascript in web pages and macros embedded in word processing documents, tends to blur the code/data distinction and create new risks.

7.4 Generality

Although our paper illustrates the idea of randomizing instruction sets at the machine-code level, the basic concept could be applied wherever it is possible to (1) distinguish code from data, (2) identify all sources of trusted code, and (3) introduce hidden diversity into all and only the trusted code. A RISE for protecting `printf` format strings, for example, might rely on compile-time detection of legitimate format strings, which might either be randomized upon detection, or flagged by the compiler for randomization sometime closer to runtime. Certainly, it is essential that a running program interact with external information, at some point, or no externally useful computation can be performed. However, the recent SQL attacks illustrate the increasing danger of expressing running programs in externally known languages [Harper 2002]. Randomized instruction set emulators are one step toward reducing that risk.

An attraction of RISE, compared to an approach such as code shepherding, is that injected code is stopped by an inherent property of the system, without requiring any explicit or manually defined checks before execution. Although divorcing policy from mechanism (as in code shepherding) is a valid design principle in general, complex user-specified policies are more error prone than simple mechanisms that hard-code a well-understood policy.

8. CONCLUSIONS

In this paper we introduced the concept of a randomized instruction set emulator as a defense against binary code injection attacks. We demonstrated the feasibility and utility of this concept with a proof-of-concept implementation based on Valgrind. Our implementation successfully scrambles binary

code at load time, unscrambles it instruction-by-instruction during instruction fetch, and executes the unscrambled code correctly. The implementation was successfully tested on several code-injection attacks, some real and some synthesized, which exhibit common injection techniques.

We also addressed the question of RISE safety—how likely are random byte sequences to cause damage if executed. We addressed this question both experimentally and theoretically and conclude that there is an extremely low probability that executing a sequence of random bytes would cause real damage (say by executing a system call). However, there is a slight probability that such a random sequence might escape into an infinite loop or valid code. This risk is much lower for the Power PC instruction set than it is for the IA32, due to the density of the IA32 instruction set. We thus conclude that a RISE approach would be even more successful on the Power PC architecture than it is on the IA32.

As the complexity of systems grows, and 100% provable overall system security seems an ever more distant goal, the principle of diversity suggests that having a variety of defensive techniques based on different mechanisms with different properties stands to provide increased robustness, even if the techniques address partially or completely overlapping threats. Exploiting the idea that it is hard to get much done when you do not know the language, RISE is another technique in the defender's arsenal against binary code injection attacks.

APPENDIX

A. ENCODING OF THE IA32 MARKOV CHAIN MODEL

In this appendix, we discuss the details for the construction of the Markov chain representing the state of the processor as each byte is interpreted.

If $X_t = j$ is the event of being in state j at time t (in our case, at the reading of byte t), the transition probability $P\{X_{t+1} = j | X_t = i\}$ is denoted p_{ij} and is the probability that the system will be in state j at byte $t + 1$ if it is in state i for byte t.

For example, when the random sequence starts (in state $start$), there is some probability p that the first byte will correspond to an existing 1-byte opcode that requires an additional byte to specify memory addressing (the Mod-Reg-R/M (MRM) byte). Consequently, we create a transition from *start* to *mrm* with some probability p: $p_{start,mrm} = p$. p is the number of instructions with one opcode that require the MRM byte, divided by the total number of possibilities for the first byte (256). In IA32 there are 41 such instructions, so $p_{start,mrm} = \frac{41}{256}$.

If the byte corresponds to the first byte of a 2-byte instruction, we transition to an intermediate state that represents the second byte of that family of instructions, and so on. There are two exit states: *crash* and *escape*. The *crash* state is reached when an illegal byte is read, or there is an attempt to use invalid memory, for an operation or a jump. The second exit state, *escape*, is reached probabilistically when a legitimate jump is executed. This is related to the escape event.

Because of the complexity of the IA32 instruction set, we simplified in some places. As far as possible, we adhered to the worst-case principle, in which we overestimated the bad outcomes when uncertainty existed (e.g., finding a legal instruction, executing a privileged instruction, or jumping). The next few paragraphs describe these simplifications.

We made two simplifications related to instructions. The IA32 has instruction modifiers called prefixes that can generate complicated behaviors when used with the rest of the instruction set. We simplified by treating all of them as independent instructions of length 1 byte, with no effect on the following instructions. This choice overestimates the probability of executing those instructions, as some combinations of prefixes are not allowed, others significantly restrict the kind of instructions that can follow, or make the addresses or operands smaller. In the case of regular instructions that require longer low-probability pathways, we combined them into similar patterns. Privileged instructions are assumed to fail with probability of 1.0 because we assume that the RISE-protected process is running at user level.

In the case of conditional branches, we assess the probability that the branch will be taken, using the combination of flag bits required for the particular instruction. For example, if the branch requires that two flags have a given value (0 or 1), the probability of taking the branch is set to 0.25 . A nontaken branch transitions to the *start* state as a linear instruction. All conditional branches in IA32 use *relative* (to the current Instruction Pointer), 8- or 16-bit displacements. Given that the attack had to be in an executable area to start with, this means that it is likely that the jump will execute. Consequently, for conditional branches we transition to *escape* with probability 1. This is consistent with the observed behavior of successful jumps.

A.1 Definition of Loose and Strict Criteria of Escape

Given that the definition of escape is relative to the position of the instruction in the exploit area, it is necessary to arbitrarily decide if to classify an incomplete interpretation as an escape or as a crash. This is the origin of the loose and strict criteria.

In terms of the Markov chain, the loose and strict classifications are defined as follows:

(1) *Loose escape*: Starting from the *start* state, reach any state except *crash*, in m transitions (reading m bytes).
(2) *Strict escape*: Reach the *escape* state in m or fewer transitions from the *start* state (in m bytes).

If T is the transition matrix representing the IA32 Markov chain, then to find the probability of escape from a sequence of m random bytes, we need to determine if the chain is in state *start* or *escape* (the strict criterion) or not in state *crash* (the loose criterion) after advancing m bytes. These probabilities

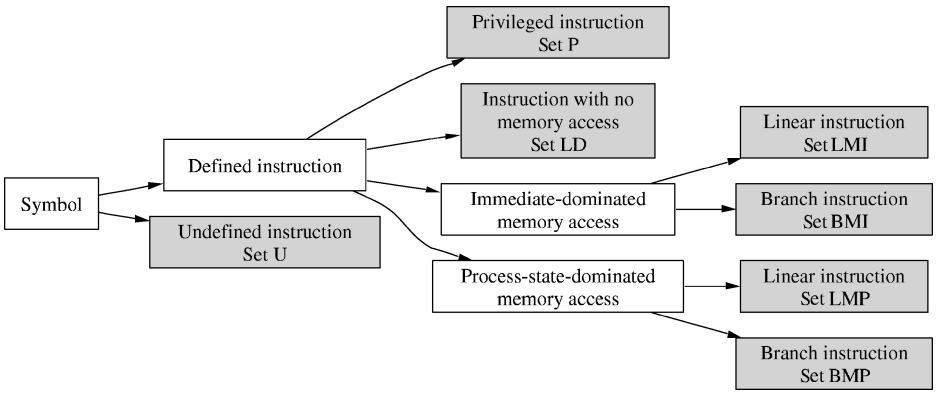

Fig. 7. Partition of symbols into disjoint sets based on the possible outcome paths of interest in the decoding and execution of a symbol. Each path defines a set. Each shaded leaf represents one (disjoint) set, with the set name noted in the box.

are given by $T^m(start,start) + T^m(start,escape)$ and $1 - T^m(start,crash)$, respectively, where $T(i, j)$ is the probability of a transition from state i to state j.

B. ENCODING OF A UNIFORM-LENGTH INSTRUCTION SET

This appendix contains intermediate derivations for the uniform-length instruction set model.

B.1 Partition Graph

Figure 7 illustrates the partition of the symbols into disjoint sets using the execution model given in Section 4.1.

B.2 Encoding Conventions

The set of branches that are relative to the current instruction pointer with a small offset (defined as being less or equal than 2^{b-1}) is separated from the rest of the branches, because their likelihood of execution is very high. In the analysis we set their execution probability to 1, which is consistent with observed behavior.

A fraction of the conditional branches are artificially separated into L_{MI} and L_{MP} from their original B_{MI} and B_{MP} sets. This fraction corresponds to the probability of taking the branch, which we assume is 0.5. This is similar to the IA32 case, where we assumed that a non-branch-taking instruction could be treated as a linear instruction.

To determine the probability that a symbol falls into one of the partitions, we need to enumerate all symbols in the instruction set. For accounting purposes, when parts of addresses and/or immediate (constant) operands are encoded inside the instruction, each possible instantiation of these data fields is counted as a different instruction. For example, if the instruction "XYZ" has 2 bits specifying one of four registers, we count four different XYZ instructions, one for each register encoding.

B.3 Derivation of the Probability of a Successful Branch (Escape) Out of a Sequence of n Random Bytes

$$\begin{aligned} P(X_n^\star) &= \sum_{i=1,\ldots,n} P(X_i) + P(L)^n \\ &= \sum_{i=1,\ldots,n} P(L)^i P(E) + P(L)^n \\ &= \left(P(E) \sum_{i=1,\ldots,n} P(L)^i \right) + P(L)^n \\ &= P(E)\frac{1 - P(L)^{n+1}}{1 - P(L)} + P(L)^n. \end{aligned} \tag{1}$$

B.4 Derivation of the Lower Limit for the Probability of Escape

$$\begin{aligned} \lim_{n\to\infty} P(X_n^\star) &= \lim_{n\to\infty} P(E)\frac{1 - P(L)^{n+1}}{1 - P(L)} + P(L)^n \\ &= \frac{P(E)}{1 - P(L)}. \end{aligned} \tag{2}$$

REFERENCES

ANDERSON, R. 2003. "Trusted Computing" and competition policy—Issues for computing professionals. *Upgrade IV*, 3 (June), 35–41.

ARBAUGH, W. A. 2002. Improving the TCPA specification. *IEEE Comput. 35*, 8 (Aug.), 77–79.

AVIJIT, K., GUPTA, P., AND GUPTA, D. 2004. Tied, libsafeplus: Tools for dynamic buffer overflow protection. In *Proceeding of the 13th USENIX Security Symposium*. San Diego, CA.

AVIZIENIS, A. 1995. The methodology of N-version programming. In *Software Fault Tolerance*, M. Lyu, Ed. Wiley, New York, 23–46.

AVIZIENIS, A. AND CHEN, L. 1977. On the implementation of N-Version programming for software fault tolerance during execution. In *Proceedings of IEEE COMPSAC 77*. 149–155.

BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming language design and implementation*. ACM Press, Vancouver, British Columbia, Canada, 1–12.

BARATLOO, A., SINGH, N., AND TSAI, T. 2000. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, Berkeley, CA. 251–262.

BARRANTES, E. G., ACKLEY, D., FORREST, S., PALMER, T., STEFANOVIC, D., AND ZOVI, D. D. 2003. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, Washington, DC. 272–280.

BEST, R. M. 1979. Microprocessor for executing enciphered programs, U.S. Patent no. 4 168 396.

BEST, R. M. 1980. Preventing software piracy with crypto-microprocessors. In *Proceedings of the IEEE Spring COMPCON '80*, San Francisco, CA. 466–469.

BHATKAR, S., DUVARNEY, D., AND SEKAR, R. 2003. Address obfuscation: An approach to combat buffer overflows, format-string attacks and more. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC. 105–120.

BOYD, S. W. AND KEROMYTIS, A. D. 2004. SQLrand: Preventing SQL injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*. Yellow Mountain, China. 292–302.

BRUENING, D., AMARASINGHE, S., AND DUESTERWALD, E. 2001. Design and implementation of a dynamic optimization framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*.

BUTLER, T. R. 2004. Bochs. http://bochs.sourceforge.net/.

CHEW, M. AND SONG, D. 2002. *Mitigating Buffer Overflows by Operating System Randomization*. Tech. Rep. CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University.

CHIUEH, T. AND HSU, F.-H. 2001. Rad: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, AZ. 409–420.

COHEN, F. 1993. Operating system protection through program evolution. *Computers and Security 12*, 6 (Oct.), 565–584.

CORE SECURITY. 2004. CORE security technologies. http://www1.corest.com/home/home.php.

COWAN, C., BARRINGER, M., BEATTIE, S., AND KROAH-HARTMAN, G. 2001. Format guard: Automatic protection from `printf` format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington, DC. 191–199.

COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. 2003. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC. 91–104.

COWAN, C., HINTON, H., PU, C., AND WALPOLE, J. 2000. A cracker patch choice: An analysis of post hoc security techniques. In *National Information Systems Security Conference (NISSC)*, Baltimore MD.

COWAN, C., PU, C., MAIER, D., HINTON, H., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. 1998. Automatic detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX.

COWAN, C., WAGLE, P., PU, C., BEATTIE, S., AND WALPOLE, J. 2000b. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*. 119–129.

DALLAS SEMICONDUCTOR. 1999. DS5002FP secure microprocessor chip. http://pdfserv.maxim-ic.com/en/ds/DS5002FP.pdf.

DOR, N., RODEH, M., AND SAGIV, M. 2003. CSSV: Towards a realistic tool for statically detecting all buffer overflows in c. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. 155–167.

ETOH, H. AND YODA, K. 2000. Protecting from stack-smashing attacks. Web publishing, IBM Research Division, Tokyo Research Laboratory, http://www.trl.ibm.com/projects/security/ssp/main.html. June 19.

ETOH, H. AND YODA, K. 2001. Propolice: Improved stack smashing attack detection. *IPSJ SIGNotes Computer Security (CSEC) 14* (Oct. 26).

FAYOLLE, P.-A. AND GLAUME, V. 2002. A buffer overflow study, attacks & defenses. Web publishing, ENSEIRB, http://www.wntrmute.com/docs/bufferoverflow/report.html.

FORREST, S., SOMAYAJI, A., AND ACKLEY, D. 1997. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*. 67–72.

FRANTZEN, M. AND SHUEY, M. 2001. Stackghost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*. Washington, DC.

GERA AND RIQ. 2002. Smashing the stack for fun and profit. *Phrack 59*, 11 (July 28).

HARPER, M. 2002. SQL injection attacks—Are you safe? In Sitepoint, http://www.sitepoint.com/article/794.

IBM. 2003. *PowerPC Microprocessor Family: Programming Environments Manual for 64 and 32-Bit Microprocessors. Version 2.0*. Number order nos. 253665, 253666, 253667, 253668.

INTEL CORPORATION. 2004. The IA-32 Intel Architecture Software Developer's Manual. Number order nos. 253665, 253666, 253667, 253668.

JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. 2002. Cyclone: A safe dialect of c. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA. 275–288.

JONES, R. W. M. AND KELLY, P. H. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In *3rd International Workshop on Automated Debugging*. 13–26.

KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*. ACM Press, Washington, DC. 272–280.

KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. 2002. Secure execution via program sheperding. In *Proceeding of the 11th USENIX Security Symposium*, San Francisco, CA.

KLAIBER, A. 2000. The technology behind the crusoe processors. White Paper http://www.transmeta.com/pdf/white_papers/paper_aklaiber_19jan00.pdf. January.

KUHN, M. 1997. *The TrustNo 1 Cryptoprocessor Concept*. Tech. Rep. CS555 Report, Purdue University. April 04.

LAROCHELLE, D. AND EVANS, D. 2001. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington, DC. 177–190.

LHEE, K. AND CHAPIN, S. J. 2002. Type-assisted dynamic buffer overflow detection. In *Proceeding of the 11th USENIX Security Symposium*, San Francisco, CA. 81–88.

MILENKOVIĆ, M., MILENCOVIĆ, A., AND JOVANOV, E. 2004. A framework for trusted instruction execution via basic block signature verification. In *Proceedings of the 42nd Annual Southeast Regional Conference (ACM SE'04)*. ACM Press, Huntsville, AL. 191–196.

NAHUM, E. M. 2002. Deconstructing specweb99. In *Proceedings of 7th International Workshop on Web Content Caching and Distribution*, Boulder, CO.

NEBENZAHL, D. AND WOOL, A. 2004. Install-time vaccination of Windows executables to defend against stack smashing attacks. In *Proceedings of the 19th IFIP International Information Security Conference*. Kluwer, Toulouse, France, 225–240.

NECULA, G. C., MCPEAK, S., AND WEIMER, W. 2002. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the Symposium on Principles of Programming Languages*. 128–139.

NERGAL. 2001. The advanced return-into-lib(c) exploits. *Phrack 58*, 4 (Dec.).

NETHERCOTE, N. AND SEWARD, J. 2003. Valgrind: A program supervision framework. In *Electronic Notes in Theoretical Computer Science*, O. Sokolsky and M. Viswanathan, Eds. Vol. 89. Elsevier, Amsterdam.

NEWSHAM, T. 2000. Format string attacks. http://www.securityfocus.com/archive/1/81565.

PAX TEAM. 2003. Documentation for the PaX project. See Homepage of The PaX Team. http://pax.grsecurity.net/docs/index.html.

PRASAD, M. AND CHIUEH, T. 2003. A binary rewriting defense against stack based overflow attacks. In *Proceedings of the USENIX 2003 Annual Technical Conference*, San Antonio, TX.

PU, C., BLACK, A., COWAN, C., AND WALPOLE, J. 1996. A specialization toolkit to increase the diversity of operating systems. In *Proceedings of the 1996 ICMAS Workshop on Immunity-Based Systems*, Nara, Japan.

RANDELL, B. 1975. System structure for software fault tolerance. *IEEE Trans. Software Eng. 1*, 2, 220–232.

RUWASE, O. AND LAM, M. S. 2004. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*.

SCHNEIER, B. 1996. *Applied Cryptography*. Wiley, New York.

SECURITY FOCUS. 2003. CVS directory request double free heap corruption vulnerability. http://www.securityfocus.com/bid/6650.

SEWARD, J. AND NETHERCOTE, N. 2004. Valgrind, an open-source memory debugger for x86-GNU/Linux. http://valgrind.kde.org/.

SIMON, I. 2001. A comparative analysis of methods of defense against buffer overflow attacks. Web publishing, California State University, Hayward, http://www.mcs.csuhayward.edu/simon/security/boflo.html. January 31.

SPEC INC. 1999. *Specweb99*. Tech. Rep. SPECweb99_Design_062999.html, SPEC Inc. June 29.

TCPA 2004. TCPA trusted computing platform alliance. http://www.trustedcomputing.org/home.

TOOL INTERFACE STANDARDS COMMITTEE. 1995. *Executable and Linking Format (ELF)*. Tool Interface Standards Committee.

TSAI, T. AND SINGH, N. 2001. Libsafe 2.0: Detection of format string vulnerability exploits. White Paper Version 3-21-01, Avaya Labs, Avaya Inc. February 6.

TSO, T. 1998. random.C: A strong random number generator. http://www.linuxsecurity.com/ feature_stories/random.c.

VENDICATOR. 2000. StackShield: A stack smashing technique protection tool for Linux. http://angelfire.com/sk/stackshield.

WAGNER, D., FOSTER, J. S., BREWER, E. A., AND AIKEN, A. 2000. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, San Diego, CA. 3–17.

WILANDER, J. AND KAMKAR, M. 2003. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, San Diego, CA. 149–162.

XU, J., KALBARCZYK, Z., AND IYER, R. K. 2003. Transparent runtime randomization for security. In *Proceeding of the 22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, Florence, Italy. 26–272.

XU, J., KALBARCZYK, Z., PATEL, S., AND IYER, R. K. 2002. Architecture support for defending against buffer overflow attacks. In *2nd Workshop on Evaluating and Architecting System dependabilitY (EASY)*, San Jose, CA. http://www.crhc.uiuc.edu/EASY/.

Received May 2004; revised September 2004; accepted September 2004